

**Study and implementation of computational
methods for Differential Equations in
heterogeneous systems**

by

Asimina Vouronikoy - Eleni Zisiou

Certified by
Panagiota Tsompanopoulou
Assistant Professor
Thesis Supervisor

Certified by
Christos Antonopoulos
Assistant Professor
Thesis Supervisor

Certified by
Panayiotis Bozanis
Associate Professor
Thesis Supervisor

Study and implementation of computational methods for Differential Equations in heterogeneous systems

by

Asimina Vouronikoy - Eleni Zisiou

Submitted to the Department of Computer ,Communication and Network
Engineering
on October , 2013,

Abstract

The purpose of this thesis was to study and implement computational methods for solving PDEs arising from 1D and 2D problems in heterogeneous systems. Differential equations are of high importance and model many physical and scientific problems. After a thorough review of previous work on the field two algorithms were chosen to be implemented. The cyclic reduction algorithm and the block cyclic reduction algorithm were implemented for CPU and ported to GPU to study the performance. The CPU code was implemented in C and Matlab. The GPU code was implemented in NVIDIA CUDA. Due to numerical instability of Block Cyclic Reduction the Buneman variant was implemented. Extensive experiments were performed and their results were compared and studied. It was proved that in GPUs the Buneman algorithm shows up to 25x speedup in relation to CPU and cyclic reduction up to 8x. That concludes that GPU can be used to solve efficiently, accurately and clearly faster these problems especially in two dimensions where the speedup was particularly remarkable and the computational load extremely large.

Thesis Supervisor: Panagiota Tsompanopoulou
Title: Assistant Professor

Thesis Supervisor: Christos Antonopoulos
Title: Assistant Professor

Thesis Supervisor: Panayiotis Bozanis
Title: Associate Professor

Acknowledgments

In this section we would like to sincerely thank our supervisors Professor Panagiota Tsompanopoulou and Professor Christos Antonopoulos for their continuous help and guidance through the development of this thesis. Also we would like to thank Professor Panayiotis Bozanis for participating in this effort. Finally, we would like to thank our families and friends for their constant support during these months.

To our Families and our Friends

Contents

1	Introduction	15
2	GPUs and Parallel Programming Models	17
2.1	Graphics Processing Units (GPU) Computing	17
2.2	NVIDIA's Kepler Architecture	19
2.3	NVIDIA CUDA Programming Model	23
3	A Review of Solving Partial Differential Equations (PDEs)	27
3.1	About PDEs	27
3.1.1	Finite Difference Methods for Solving Elliptic PDEs	28
3.1.2	Finite Difference Methods for Solving Parabolic PDE's	29
3.2	Stencil Computation	29
3.3	Solvers for PDEs	35
3.4	Conclusions of Review	49
4	Implementation	53
4.1	Cyclic Reduction	53
4.1.1	The algorithm	53
4.1.2	Implementation Issues	55
4.2	Block Cyclic Reduction	59
4.2.1	The algorithm	59
4.2.2	Implementation Issues	61

5	Results	65
5.1	Hardware and Metrics	65
5.2	Experiments	65
5.2.1	Single Precision Performance	66
5.2.2	Double Precision Performance	72
5.2.3	Conclusions	79
5.2.4	Further Improvements	80
A		81

List of Figures

2-1	GPU Architecture	18
2-2	Kepler Architecture	20
2-3	Warp Scheduler	21
2-4	Kepler Memory Hierarchy	22
2-5	Cuda Blocks	24
2-6	Cuda Grids	25
3-1	Discretization techniques	49
4-1	CR Method	55
5-1	CPU-GPU Cyclic Reduction(SP)	66
5-2	Speedup Cyclic Reduction(SP)	67
5-3	Speedup Buneman(SP)	71
5-4	Speedup Buneman N=Q (SP)	72
5-5	CPU-GPU Cyclic Reduction(DP)	73
5-6	Speedup Cyclic Reduction(DP)	74
5-7	Speedup Buneman(DP)	78
5-8	Speedup Buneman N=Q (DP)	79

List of Tables

3.1	Classification of PDEs	28
3.2	Conclusion Table	51
5.1	Cyclic Reduction (first version) execution time (SP)	66
5.2	Cyclic Reduction (first version) speed - up (SP)	67
5.3	Cyclic Reduction (padding version) execution time (SP)	68
5.4	Cyclic Reduction (padding version) speed - up (SP)	68
5.5	Block Cyclic Reduction N=3	69
5.6	Block Cyclic Reduction N=7	69
5.7	Block Cyclic Reduction N=15	69
5.8	Block Cyclic Reduction N=31	70
5.9	Block Cyclic Reduction N=63	70
5.10	Block Cyclic Reduction N=127	70
5.11	Block Cyclic Reduction N=Q	71
5.12	Cyclic Reduction (first version) execution time (DP)	72
5.13	Cyclic Reduction (first version) speed - up (DP)	73
5.14	Cyclic Reduction (padding version) execution time (DP)	74
5.15	Cyclic Reduction (padding version) speed - up (DP)	75
5.16	Block Cyclic Reduction N=3	76
5.17	Block Cyclic Reduction N=7	76
5.18	Block Cyclic Reduction N=15	76
5.19	Block Cyclic Reduction N=31	77
5.20	Block Cyclic Reduction N=63	77

5.21	Block Cyclic Reduction $N=127$	77
5.22	Block Cyclic Reduction $N=Q$	78

Chapter 1

Introduction

Tridiagonal solvers are tools of high importance in wide range of engineering and scientific applications. Some applications of tridiagonal solvers include computer graphics, financial applications, fluid dynamics, Poisson solvers, preconditioner in iterative solvers and cubic spline calculation. Recent technological evolution of GPUs has lifted many scientific and engineering applications to a level that was only possible with room-sized supercomputers in the past.

There are various methods solving tridiagonal systems such as Thomas algorithm, cyclic reduction method, recursive doubling etc. Some of them exploit parallelism and a lot of research has been made over the years in order to improve their performance.

This thesis is organized as follows. Chapter two provides an introduction to GPU computing and all the necessary background information in NVIDIA's CUDA programming model. Chapter three is a review of solving partial differential equation on GPUs is presented. Specifically this Chapter gives a summary of the related work that is done in the field of stencil computation and PDEs solving on GPUs. Chapter four describes the implementations of and optimizations. Two methods were implemented and ported on GPU, cyclic reduction (CR) that solves tridiagonal linear systems and block cyclic reduction (BCR) that solves block tridiagonal linear systems. In the last Chapter, performance comparison, results of the experiments performed and the concluding remarks are demonstrated.

Chapter 2

GPUs and Parallel Programming Models

2.1 Graphics Processing Units (GPU) Computing

Conventional central processing units (CPU) are reaching the limits for how high the clock frequencies can go. More and more cores are added on each CPU so as to achieve higher computational throughput. However, physical limitations and limitations on the circuit fabrication make more difficult the improvement of the clock speed of the CPU.

Therefore, the CPU faces the problem of keeping its clock frequency growing and have added more cores to counterbalance this. The other hardware part that is used, the graphical processing units (GPUs), which have always been parallel hardware made for real-time 3D renderings, have evolved into devices that may also be programmed and used for general-purpose computing like scientific computations. Specifically, GPUs can run many threads in parallel that fulfill high computation demand and provide large memory bandwidth to serve parallel memory access requests.

Hence, parallel computing imported a new area called GPGPU, or General-Purpose computation on GPU. The importance of GPGPU technology is to provide heterogeneous computations where applications use both the CPU and GPU. Simply, it can increase the speed of applications with a large amount of data just by using the

GPU as a co-processor to CPU to accelerate its general purpose computations that were once only managed by the CPU alone. NVIDIA and ATI have been the main GPU manufacturers with a long list of different models and features. Both of these companies have been producing different platforms that can use parallel computing architectures to utilize the GPU's stream processors in order to gain more speed-up for any computing process.

Nowadays, CPU systems are basically multi-core systems. They contain handful of strong cores, each supporting approximately one hardware thread. In contrary, GPU systems are called many-core systems containing many but "lighter" threads. There are two major differences between CPU and GPU threads. First, GPUs have small caches but the context switching between threads is essentially fast so as to overlap memory access latency with useful computations. And second, GPUs achieve high performance when thousands of threads execute in parallel while CPUs need the number of threads per core to be small.

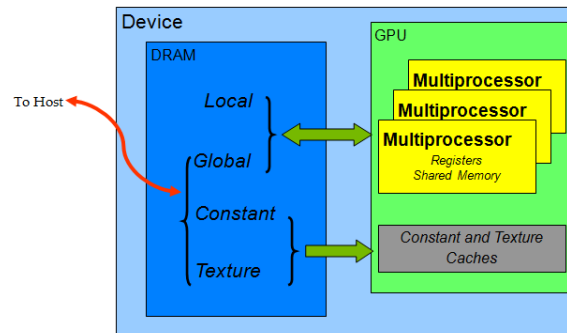


Figure 2-1: GPU Architecture

As shown in Figure 2-1, GPU is presented as a set of multiprocessors. Each multiprocessor has its own shared memory and registers. The processors connect with DRAM via an interconnection Network.

2.2 NVIDIA's Kepler Architecture

In an attempt to interpret better the results, it seems necessary to explain the GPU architecture (Kepler GK110 architecture) used in this thesis and especially the GPU model (GTX Force 680) used for the development and the experiments.

NVIDIA's Kepler architecture introduced to improve mainly 3D graphics quality to gamers. Kepler builds on the foundation first established with NVIDIA's Fermi GPU architecture. This architecture demonstrates higher performance and more efficiency in terms of power consumption compared to Fermi. Kepler provides over 1 TFlop of double precision throughput with greater than 80% DGEMM efficiency versus 60 - 65% on the prior Fermi architecture. The first product being introduced based on Kepler architecture is the GeForce GTX 680. The design of this architecture was basically focused on improving power efficiency, delivering up to 3x the performance per watt of Fermi.

In the design of Kepler introduced a new Streaming Multiprocessor, called "SMX" one of the keys to GeForce GTX 680's performance. For improved power efficiency, the SMX runs at graphics clock rather than 2x graphics clock, but with 1536 CUDA cores in GK104, the GeForce GTX 680 SMX provides 2x the performance per watt of Fermi's SM (GF110). This allows the GeForce GTX 680 to deliver higher performance/watt when compared to GeForce GTX 580. The GeForce GTX 680 GPU consists of four Graphics Processing Clusters (GPCs), eight next-generation Streaming Multiprocessors (SMX), and four memory controllers.

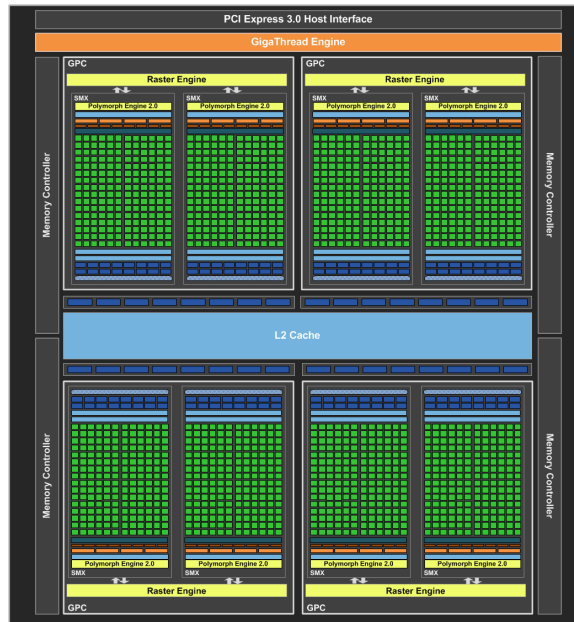


Figure 2-2: Kepler Architecture

GeForce GTX 680 contains four GPCs, delivering 32 pixels per clock. Each GPC has a dedicated raster engine and two SMX units. With a total of eight SMX units, the GeForce GTX 680 implementation has 1536 CUDA Cores. GeForce GTX 680's memory subsystem was also completely revamped, resulting in dramatically higher memory clock speeds. GeForce GTX 680 operates at 6008MHz data rate. Tied to each memory controller are 128KB L2 cache and eight Raster Operations (ROP) units (each of the eight ROP units processes a single color sample). With four memory controllers, a full GeForce GTX 680 GPU has 512KB L2 cache and 32 ROPs (i.e., 32 color samples).

The SM is the heart of NVIDIA's unified GPU architecture. Most of the key hardware units for graphics processing reside in the SM. The SM's CUDA cores perform pixel/vertex/geometry shading and physics/compute calculations. Texture units perform texture filtering and load/store units fetch and save data to memory. Special Function Units (SFUs) handle transcendental and graphics interpolation instructions. Finally, the Polymorph Engine handles vertex fetch, tessellation, viewport transform, attribute setup, and stream output. Kepler GK110 supports the CUDA Compute Capability 3.5.

SMX Processing Core Architecture each of the Kepler GK110 SMX units feature 192 single precision CUDA cores, and each core has fully pipelined floating - point and integer arithmetic logic units. Kepler retains the full compliant single and double precision arithmetic introduced in Fermi, including the fused multiply - add (FMA) operation. The SMX schedules threads in groups of 32 parallel threads called warps. Each SMX features four warp schedulers and eight instruction dispatch units, allowing four warps to be issued and executed concurrently. Kepler's quad warp scheduler selects four warps, and two independent instructions per warp can be dispatched each cycle.

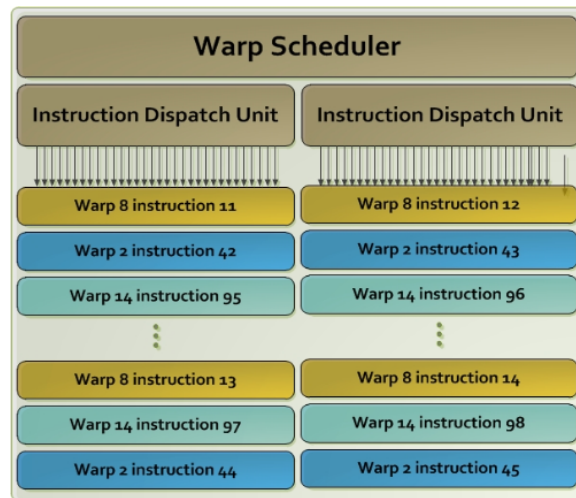


Figure 2-3: Warp Scheduler

To improve performance Kepler introduced a new shuffle instruction, which allows threads within a warp to share data. Previously, sharing data between threads within a warp required separate store and load operations to pass the data through shared memory. With the Shuffle instruction, threads within a warp can read values from other threads in a warp in just about any imaginable permutation. Shuffle supports arbitrary indexed references.

In order to increase performance changes have been made in Kepler for the atomic operations. The atomic operations are executed individually from every thread and their execution cannot be interrupted. Atomic operations are widely used in parallel programming in order to succeed synchronization between threads. Throughput of

global memory atomic operations on Kepler GK110 is substantially improved compared to the Fermi generation. Atomic operation throughput to a common global memory address is improved by 9x to one operation per clock. Kepler GK110 also expands the native support for 64-bit atomic operations in global memory. Besides atomicAdd, atomicCAS, and atomicExch, GK110 supports atomicMax, atomicMin, atomicAnd, atomicOr and atomicXor [1].

Kepler keeps a similar memory hierarchy to Fermi. The Kepler architecture supports a unified memory request path for loads and stores, with an L1 cache per SMX multiprocessor. Kepler GK110 also enables compiler directed use of an additional new cache for read-only data as shown in Figure. As in Fermi architecture in the Kepler GK110 architecture, each SMX has 64 KB of on-chip memory that can be configured as 48 KB of shared memory with 16 KB of L1 cache, or as 16 KB of shared memory with 48 KB of L1 cache. In Kepler now is permitted a 32KB/32KB split between shared memory and L1 cache, something that offers extra flexibility. To support the increased throughput of each SMX unit, the shared memory bandwidth for 64b and larger load operations is also doubled compared to the Fermi SM, to 256B per core clock.

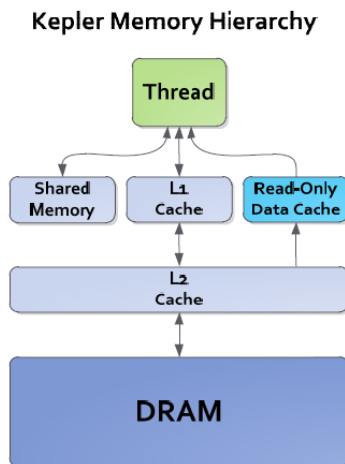


Figure 2-4: Kepler Memory Hierarchy

Except from L1 cache, an 48KB cache for data is added that is known to be read-only for the duration of the function. Another feature met in Kepler GK110

GPU is an improved L2 cache where there are 1536KB of dedicated L2 cache memory, double the amount of L2 available in the Fermi architecture. The L2 cache on Kepler offers up to 2x of the bandwidth per clock available in Fermi. Like Fermi, Kepler's register files, shared memories, L1 cache, L2 cache and DRAM memory are protected by a Single-Error Correct Double-Error Detect (SECCDED) ECC code. In addition, the Read-Only Data Cache supports single - error correction through a parity check; in the event of a parity error, the cache unit automatically invalidates the failed line, forcing a read of the correct data from L2.

2.3 NVIDIA CUDA Programming Model

Compute Unified Device Architecture (CUDA) is a general purpose parallel computing platform and programming model introduced by NVIDIA. CUDA is mostly used to increase the efficiency in the solution of large computational problems in relation to CPU.

CUDA C is an extension of C. Actually CUDA can be considered as C with a few keywords. The programmer defines functions, called kernels. Kernels are executed N times in parallel by N different CUDA threads and only access GPU memory. A kernel can be defined with the `__global__` declaration specifier when launched from CPU. A kernel may be launched from another kernel using the `__device__` declaration specifier. The programmer is able to define the number of CUDA threads that execute the kernel using the `<<...>>` execution configuration syntax. At each thread that executes the kernel is assigned a unique thread ID that can be referred to via the `threadIdx` variable. Threads are managed in groups of 32, called warps. Instructions are issued per warp. If an operand is not ready the warp will stall. Computations can be performed in one, two and three-dimensions. For that purpose, `threadIdx` is a 3-component vector (`threadIdx.x`, `threadIdx.y`, `threadIdx.z`). This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume. The number of threads per block is limited, due to limited memory resources of the processor core, on which all threads per block within the block reside.

In most cases the number of threads per block reflects the problem geometry. On current GPUs, threads per block are up to 1024 threads.

Blocks do not migrate among processors, execute on one multiprocessor. Several blocks can execute concurrently on one multiprocessor. They are organized into a one, two, or three-dimensional grid of thread blocks as illustrated by Figure 2-5. The number of thread blocks in the grid usually results from the amount of data to be processed to the number of threads per block.

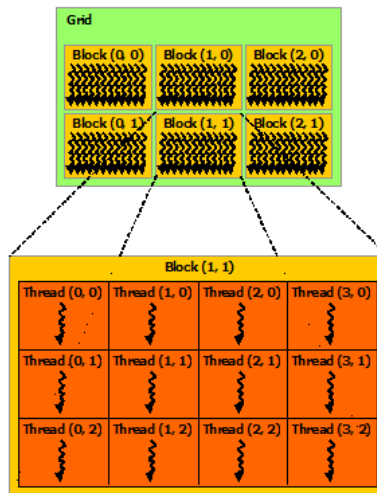


Figure 2-5: Cuda Blocks

In correspondence with the threads in the blocks, each block within the grid can be identified by a one, two, or three-dimensional index accessible within the kernel through the built-in blockIdx variable. The dimension of the thread block can be accessed within the kernel through the built-in blockDim variable.

Every thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. Shared memory is used by threads to contribute for shared data and synchronize their execution to avoid simultaneous memory accesses. In particular, by calling the `__syncthreads()` function threads can succeed synchronization. `__syncthreads()` is a barrier where all threads in a block must wait before any is allowed to proceed. All threads have access to the same global memory. The constant and texture memory spaces are two additional read-only memory spaces accessible by all threads. The global, constant, and

texture memory spaces are persistent across kernel launches by the same application.

In correspondence with the threads in the blocks, each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional index accessible within the kernel through the built-in `blockIdx` variable. The dimension of the thread block can be accessed within the kernel through the built-in `blockDim` variable. Every thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. Shared memory is used from threads to cooperate, share data and synchronize their execution to avoid simultaneous memory accesses. In particular, by calling the `__syncthreads()` function threads can succeed synchronization. `__syncthreads()` is a barrier where all threads in a block must wait before any is allowed to proceed. All threads have access to the same global memory. The constant and texture memory spaces are two additional read-only memory spaces accessible by all threads. The global, constant, and texture memory spaces are persistent across kernel launches by the same application ([2], [3], [4]).

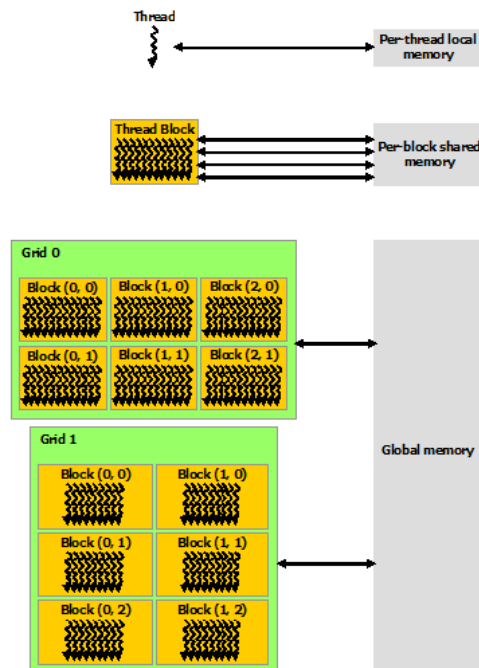


Figure 2-6: Cuda Grids

The CUDA programming model assumes a system composed of a host and a de-

vice, each with their own separate memory. Serial code executes in a host (CPU) thread. Parallel code executes in many device (GPU) threads across multiple processing elements. Device memory is typically allocated using `cudaMalloc()` and freed using `cudaFree()` and data transfer between host memory and device memory are typically done using `cudaMemcpy()`.

Here are some examples of typical instructions:

Kernel definition example:

```
__global__ void kernel( int *a, int dimx, int dimy ){
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int ty = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = ty*dimx + tx;
    a[idx] = a[idx]+1;
}
```

Kernel launch example:

```
int *d_a = 0;
cudaMalloc ( (void**)&d_a, num_bytes );
dim3 grid, block;
block.x = 4;
block.y = 4;
grid.x = dimx / block.x;
grid.y = dimy / block.y;
kernel<<<grid, block>>>( d_a, dimx, dimy );
```

Chapter 3

A Review of Solving Partial Differential Equations (PDEs)

3.1 About PDEs

Partial differential equations (PDEs) are used to model many real world phenomena such as wave propagation, fluid flow, heat flow and distribution, etc. A linear 2nd-Order PDE of the general form is:

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Du_x + Eu_y + Fu + G = 0$$

where the coefficients A, B, C etc. may depend upon x and y.

If $(A^2 + B^2 + C^2) > 0$ over a region of the x-y plane, the PDE is second-order in that region. PDEs can be classified into elliptic, parabolic and hyperbolic equations. The classification is based in the discriminant $B^2 - 4AC$ as shown in Table 3.1. Parabolic equations, with the thermal conductivity equation as the most characteristic example, involve time dependency and describe diffusion. Hyperbolic equations, with the wave equation as the most standard example, are time dependent and describe dissemination phenomena. Unlike the previous, elliptic equations describe the static behavior of a magnitude in a particular area without time dependency. The most characteristic elliptical equation is the Laplace equation. Parabolic and hyperbolic equations are usually defined as initial value problems and elliptic equations as boundary value problems.

Discriminant	Type
$B^2 - 4AC < 0$	Elliptic (e.g. Laplace Eq.)
$B^2 - 4AC = 0$	Parabolic (e.g. Heat Eq.)
$B^2 - 4AC > 0$	Hyperbolic (e.g. Wave Eq.)

Table 3.1: Classification of PDEs

Most frequently, PDEs are solved using Finite Difference (FD), Finite Element Methods (FEM) and Finite Volume (FV) methods. The Finite Difference technique is implemented and mathematically analyzed more easily than the other techniques, since they formulated for structured meshes/grids. The least popular is the Finite Volume approximation because it is only two decades scientists are working on them but they are highly known further for high accuracy results in both structured and unstructured meshes.

3.1.1 Finite Difference Methods for Solving Elliptic PDEs

The first step concerns the discretization of the PDE domain into a grid of evenly spaced points (nodes) and the restriction of the PDE equation on these nodes. The second step involves the discretization of the partial derivatives and possibly its boundary conditions for every grid node. For the 2-D simple PDE problem

$$u_{xx} + u_{yy} + u = f \text{ in } \Omega, u = g \text{ on } \partial\Omega$$

and if the (1-21) stencil for the discretization of the second derivative is used, then the continuous equation becomes the following system of linear equations:

$$\frac{\partial^2 u}{\partial^2 x} + \frac{\partial^2 u}{\partial^2 y} = \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2}$$

The scheme used in this example is Finite Difference scheme, since it considers a central node and four neighbors. If the accuracy of the results is of major importance, Finite Difference schemes with accuracy higher than second order can be used. These equations involve more than four neighbor nodes, e.g., the nine point scheme. The extension of this method in 3-D can be done easily, increasing the number of the nodes

involved in the equation. In this case the equations involve seven and twenty-seven neighbor nodes for second or higher order accuracy, respectively.

The final step concerns the solution of the resulting linear system. The linear algebra methods for the solution are divided in two basic categories, Iterative (Jacobi, Gauss-Seidel, SOR, Multigrid etc) and direct methods (Gauss, Cholesky, Thomas, FFT etc).

3.1.2 Finite Difference Methods for Solving Parabolic PDE's

The first step, as before concerns the discretization of the domain into grid of evenly spaced points (nodes). The second step involves the expression of the derivatives in terms of Finite Difference Approximations of $O(h^2)$ and $O(t)$ [or $O(t^2)$ order] :

$$\frac{\partial^2 T}{\partial^2 x}, \frac{\partial^2 T}{\partial^2 y}, \frac{\partial T}{\partial t} \implies \textit{FiniteDifferences}$$

As a final step, remains the choice of $h = x = y$, and t and use of the initial and boundary conditions to solve the problem by systematically moving ahead in time. For the time derivative referred above, explicit (Euler, Leapfrog, Runge-Kutta) and implicit (Crank-Nicolson Method, ADI methods, etc.) schemes are used. Explicit methods express all future values ($t + t$) in terms of current (t) and previous ($t - t$) information, which are known. Implicit methods derive future values ($t + t$) by solving appropriate equations in terms of current (t) and sometimes previous ($t - t$) information.

3.2 Stencil Computation

As shown above, Finite Difference methods are mainly used as the second step of the PDE solution. These methods perform nearest neighbor computations called stencils. In stencil computations, each point of the grid is updated depending on a subset of its neighbors. In this way, the coefficients of the PDE are represented. Excessive work is done in this field aiming to achieve efficiency, optimization and high performance on GPUs.

Brandvik and Pullan [5] present a generalized software framework named SBLOCK

for applications that use stencil computations to solve PDEs. They used the 3D heat diffusion equation as a model problem to demonstrate their results. A second order center Finite Difference approximation scheme is used for the derivatives and the Jacobi iterative method is used for the solution of the derived system of the linear equation. According to the authors their main contribution is the combination of the automatic source code generation with a run-time library. The run-time library provides an API that is used by the application for different functions and is divided in host library and device library. The application interacts with the run-time library either through the host library or by making calls directly to the device library. This host-device abstraction is useful since only a new device library has to be written to support a new type of processor, the host library always remains the same. One of the problems is the performance bottleneck caused by the transfer of the data across the PCI-Express bus. This can be solved in different ways, depending on the problem. If the boundary conditions can be expressed in the form of a SpMV, the application can use SBLOCK's SpMV library directly. Therefore no data will be transferred across the PCI-E bus unless it needs information from another processor. If the boundary conditions are expressed differently the application may request a pointer to the array directly from the device library, and implement its own functions in NVIDIA's CUDA language to operate on this array. For the second component, the source code generation, it is important to highlight that with this approach firstly they achieved to enable multiple platforms using only a single, high-level definition of the computations performed by the solver. Secondly, because of the abstract view of the kernel implementation that the developer has, SBLOCK can use any optimization strategy offering code readability. The optimization strategies which were used by the framework were about achieving high performance using domain decomposition and multiple threads. Moreover to maximize the amount of reuse of data in the on-chip memories, they used the "cyclical queue" strategy [4]. The framework runs in both multiprocessors and GPUs. For their experiments they used: Intel Core i7 920 (Nehalem), AMD Phenom II X4 940 and NVIDIA GTX 280. The kernels of the algorithm are split into two categories: stencil kernels and non-stencil kernels. After

studying the performance and the power efficiency of different important stencils, in both single and double precision, they came to the conclusion that good performance has been demonstrated for a range of stencil kernels with different stencil sizes and arithmetic intensities. They also showed that there is a good scaling across a large GPU cluster (good results up to 16 GPUs and scaling up to 64 GPUs).

Paulius in [6] describe a GPU parallelization of the 3D finite difference computation using CUDA. The main object of this paper is how the data access redundancy evaluates implementation efficiency for both stencil-only computation and discretization of the wave equation. In the implementation the goal was to reduce redundancy (the ratio between number of elements accessed and number of elements processed) by performing calculation from shared memory. In 2D computation the 2D tile was loaded in shared memory so as the output tile to be computed. In this paper they presented an extension to 3D with two approaches, two pass and single pass. The approach of the two pass traverses the input volume twice. The first pass concerns the computation of the 2D stencil values while during the second pass, the partial results from the first pass are combined. With this approach they referred that they had an important improvement in redundancy over the naive approach which computes an order k -stencil by re-fetching all input elements for every output value, and it becomes better with the single-pass approach. Experiments were performed on stencil-only computations and on finite difference of the wave equation. About the stencil-only computation all configurations were processed using 16×16 thread blocks operating 16×16 output tiles. The experiments concerned different volume dimensions in relation to order in space, where throughput evaluated as a metric. These experiments showed that for fixed volume dimensions, throughput decrease with increased orders, is largely due to higher read redundancy, with additional arithmetic being another contributing factor. For the 3-D finite difference of the wave equation, two kernels were implemented. About the first kernel they utilized 16×16 thread blocks and output tiles with redundancy at order of 5. For the second kernel they utilized 32×16 thread blocks, 32×32 output tiles and the redundancy was 4.5. GPU performance was roughly an order of magnitude higher than a single 4-core Harper-

town Xeon, running an optimized implementation of the same computation. It was also demonstrated a multi-GPU implementation of the wave-equation finite difference. Performance results (using 16×16 tiles and thread blocks arranged as 16×16 threads) are shown for up to 4 GPUs. It was showed an almost linear speedup in two and four GPUs except one case($544 \times 544 \times 400$ does not scale linearly with 4 GPUs due to the fact that each GPU computes only 100 slices which takes significantly less time than corresponding communication). They indicated that communication overhead is hidden as long as the number of slices per GPU is 200 or greater.

Foster [7] in his paper proposed three main optimizations for solving 2D parabolic PDEs using difference equations. The approach of the red-black array was used from the representation of the grid. As first optimization, they focused on the separation of the unified red-black array into a single red- array and a single black one. The division was performed on the CPU and the separated arrays were passed to the CUDA kernel. In this way they avoided divergence since the entire warp handled either red or black points. With the second optimization strategy they aimed to reduce the number of sequential memory accesses. For updating N points in the grid, it was required 5 sequential memory accesses. To eliminate this extra latency, they reduced the amount of work per thread block. The final optimization concerned the shared memory. Because of the utilization of data, they reduced redundant memory reads through shared memory storing which was faster than accessing RAM. The computer used for testing was Intel Core i7 920 at 2.67 GHz. The two graphics cards used for GPU computing were an NVIDIA Tesla and an EVGA GeForce 480. The four versions of the kernel tested were: CPU, GPU with no optimization, GPU-RB for the first optimization and GPU-RBS for the remaining two optimizations. Throughput was used as a metric with single floating point precision. First, the throughput of the CPU implementation declined quickly in all tests from 500 to about 1500 points per dimension and gradually tapered off more as the data set sizes continued to increase. Over the same range of 500 to 1500 points per side, the throughput of the GPU versions greatly increased in relation to the number of threads and the dimension of the problem. For both cards, the first strategy, with the division of the unified

array, showed an important speed up over the basic GPU implementation, while the GPU-RB version yielded speeds of around 50%. Finally they underlined that how advantageous these strategies are, depends on the GPU architecture and GPU hardware.

Schäfer and Fey [9] presented a number of different approaches to conduct stencil computations on GPGPUs, based on a range of micro benchmarks. They choose the Jacobi iteration as a standard benchmark because it has a low FLOP per byte ratio and is the most common method. Their research is embedded into LibGeoDecomp (a generic library which can automatically parallelize user supplied stencil codes []). According to this paper the performance of a stencil code is a result of a lot of things which have to be considered and they dedicated a section writing down how to achieve the GPGPU’s maximum memory bandwidth, hide the several hundred cycles of memory latency and saturate the arithmetic pipelines. In each sequential Jacobi iteration, each update calculates a weighted average of the old cell with its neighbors in the six cardinal directions. The general approach was to break down the total volume into smaller chunks. Each chunk was assigned to a CUDA thread block. How the cells of a chunk are distributed among the threads of a thread block depends on the algorithm. The authors developed 3 approaches: the “naïve algorithm”, the “cached plane sweep” and the “pipelined wavefront”. In the first algorithm all seven values for single cell update are loaded directly from memory. Because of the cost of calculations which have to be carried with this algorithm they suggested an improvement related to the number of threads that update cells. In the “cached plane sweep” algorithm all threads of a block were put on a 2D plane and this plane is swept through the chunk. They paid attention on optimizations related to L1 cache reuse, so as the accesses to neighbor values in the same plane to be mostly served from L1 cache and they stored cells, in the direction of the moving plane, in registers. For the last algorithm, the “pipelined wavefront”, they take advantage of the von Neumann neighborhood’s property, that neighboring cells outside of the plane’s direction of travel have to be read only from one layer. In this way they managed to reduce planes to two. Their testbed consists of an NVIDIA GeForce GTX 480 consumer GPGPU and a Tesla

C2050. All algorithms were tested with a block size of $32 \times 16 \times 1$ threads because they found out that was the best geometry for their experiments. The “naïve” as well as the “cached plane sweep” algorithm updated blocks of $32 \times 16 \times 8$ cells, while the “pipelined wavefront” algorithm turned out to run faster when updating $32 \times 32 \times 64$ cells. They concluded that the “pipelined wavefront” is the fastest algorithm on the Tesla card but only for a specific size of models because shared memory is limited and as future work, performance could be improved.

Playne and Hawick [10] in their paper presented two methods, for implementing finite difference field-equation simulations on multiple GPUs. The main issue is the reduction of communication of cell halos and the overlap communications with computations. They noticed that even if finite difference methods are used more often, first-order Euler time integration method or Runge-Kutta method are easiest. First, they deal with the determination of the size of the equation memory halo and how the equation can be split across multiple GPUs. More communication between cells in each step is required for equations with a large memory halo. The field equation used as example was the Cahn-Hilliard equation. They used the best implementation for this equation and they tried to improve this simulation by spreading the computation across two GPUs. In their paper they describe in details the basic design of the best implementation. In short texture memory is used for reading the value of the cells and the surrounding neighbors from memory. This improved significantly the performance of the simulation. For the field decomposition, the field was divided in two halves and one half is loaded into memory on each GPU and with two sets of bordering cells from the other half. The width of this model depends on the memory halo of the model. The field should be split in the highest dimension so as the cells in the borders to be in sequential addresses. The first method they present is the synchronous memory copy. For this implementation they used the pThreads library to manage the multi-threading on the CPU. The main problem was that while the CPU threads exchanged data, the each GPU stopped and sit idle. For this reason they implemented the second method, the asynchronous one. CUDA supports asynchronous host-device memory access and execution for GPUs. They used streams to split de-

vice execution so as to can take place at the same time with memory copies. This is very useful for a GPU idle time reduction. The basic idea behind this implementation was to use asynchronous copies to exchange the border cell values between the two threads while the GPU is still computing the simulation for the rest of the field. One problem worth mentioning that they faced, was that for the GPU to copy data from the device memory to the host, the host memory must be allocated by CUDA to ensure that it is page locked. This problem resolved successfully using a flag that tells the compiler to make the memory available to all CUDA contexts rather than only the one used to declare it. The GPUs used for the simulations is one NVIDIA GeForce GTX 295 which contains two GPUs. In most experiments the asynchronous memory copy implementation was shown that provide the best performance. As the field length and memory halo increasing in size the performance keep becoming better. A certainly unexpected effect was the loss in performance seen when the field lengths are not powers of two. As a conclusion, they underlined that the correct use of asynchronous memory communication can provide almost linear speed up over a single-GPU implementation for larger system sizes.

3.3 Solvers for PDEs

After discretization, elliptic equations lead to algebraic equations. Most previous works use implicit schemes to discretize the PDEs that lead to solving a sparse linear system. Many numerical methods, such as Jacobi, Gauss–Seidel, conjugate gradient, and multigrid, have been applied. Explicit methods are more expensive but as the computing power is growing, there are a lot of works which use an explicit approach and a responding PDE solver.

Daniel Egloff [11] [25] uses the parabolic convection diffusion equation $\frac{du}{dt} = L(t) * u(t, x)$ where, in case of a single risk factor, $L(t)$ is the differential operator, to present a tridiagonal solver on GPUs using cyclic reduction. The purpose of this is to improve the performance of finite difference PDE solvers on GPUs. The idea of cyclic reduction is to eliminate variables from adjacent equations and reduce the system recursively

until a single equation or a two-by two system remains. The parallel cyclic reduction algorithm is a slight variation, which applies the reduction simultaneously to all n equations. The storage requirements of the solver are three diagonal vectors l , d , u for the system and one vector h for the right hand side, which also will hold the result. The solver reuses l , d , u to store the recursively generated coefficients and therefore no additional temporary storage is required. In this implementation each thread processes one row of the matrix when the matrix is sized up to a maximum 512 (NVIDIA Tesla has at most 512 threads per block). When there is need for larger matrices one thread processes multiple rows using the C++ preprocessor to roll out a sufficiently large number of if-blocks and to have one group of temporary variables for every if-block. The solver works best if all the vectors can be stored in shared memory. The threads that do not fit in shared memory are stored in global memory. First it was compared Gaussian elimination along the lines of Forsythe and Moler and the serial version of the cyclic reduction algorithm against the SSE optimized Intel MKL solver `sgtsv` and `dgtsv` in single and double precision. The benchmark is executed on an Intel Core 2 Duo CPU T9600 at 2.8 GHz. All tests have been built with full optimization turned on. It is interesting to note that the Forsythe-Moler algorithm in double precision has approximately the same performance as `dgtsv` while the serial CR underperforms with about double execution time. For single precision MKL solver was executed in almost half time compared to serial CR, with serial CR 2.06 times slower and Forsythe-Moler 1.43 times slower in the biggest problem. Subsequently, GPU implementation was compared to CPU SSE MKL solver `sgtsv` solving up to 12000 equations. A speedup 13x was obtained. The author presents various experiments measuring the effect of shared memory use, data size etc. It is noted that these experiments were performed in an Intel Core 2 Quad CPU and NVIDIA Tesla C1060 GPU.

Daniel Egloff [12] presents the implementation of efficient GPU solver with finite difference schemes for two-factor models, with a focus on stochastic volatility models, the Hundsdorfer – Verwer scheme and the Douglas scheme. The resulting partial differential equations of two state variables are solved with alternating direction implicit

(ADI) schemes. The applied methodology for the resolution of the problem is the following: the calculations are decomposed for every time step into multiple kernel calls. The Douglas scheme requires three kernel calls per time step. The suggested implementation of the Douglas scheme exhibits parallelism at two different levels. The first level is given by the sweeping over the slices in the two coordinate axes. The second level is inside of the solution algorithm of each tridiagonal system. The implementation of the Hundsorfer–Verwer scheme is very similar but slightly more complex. Benchmark of the GPU ADI solvers fulfilled for the two-dimensional Heston stochastic volatility model against an optimized, fully multithreaded CPU implementation, for which was used the Intel threading building block library (Intel TBB Team 2010). On a recent C2050 Fermi GPU, a speedup of more than a factor of 70 attained for a sufficiently large problem size. The relative performance for the Douglas scheme in single precision on an Intel dual core E5200 2.5 GHz with a GTX260 GPU goes as follows: the GPU ADI solver runs about 40 times faster than the CPU single-core version and 27 times faster than the optimized two-core version. On a Tesla C1060 or GTX260, the GPU ADI solver can handle state grids of at most 1,004 points because of shared memory and register limitations. The best speedup is achieved when the state grid size is near this limit. For small scale problems, the speedup is not very large, due to the cost of allocating device memory. If the problem size is growing, the time required to allocate memory on the GPU becomes less dominant and the speedup increases significantly. The more accurate Hundsorfer–Verwer is about twice as intensive as the Douglas scheme, which is confirmed by the timings shown.

Won-Ki Jeong et al [13] in their paper describe a 3D, volumetric, seismic fault detection system. The main contribution of their work was a novel 3D directional anisotropic diffusion algorithm based on the orientation of the seismic strata. They also implemented an application of techniques and technologies to seismic data interpretation. In this paper, presented some previous work and was highlighted that their approach is a fully 3D implementation. They summarize their fault detection method on four steps. Step 1 concerns the structure tensor analysis. A structure tensor J is defined by a tensor of a n -dimensional vector x ($J = x \nabla x$). After the eigenanalysis of

J matrix, a new positive semi-definite matrix is resulted. Its eigenvalues and eigenvectors are found using any analytic method. During the second step, a directional anisotropic diffusion is applied so as to remove noise. Specifically they modified the coherence enhancing diffusion ($\frac{\partial I}{\partial t} = \nabla * (D \nabla I)$) and they employ the explicit Euler integration scheme to solve it. Step 3 includes the computation of a fault-likelihood volume. Finally in step 4 they proposed a hysteresis thresholding to keep only strong features or features connected to features. For hysteresis thresholding, couldn't be used recursive functions as is typical in a conventional (CPU) implementation. However in their GPU implementation, an iterative method was used, checking if any point is larger than either upper or lower threshold. They performed a system 20 times faster than the CPU implementation using a PC with a Pentium 4 3.6GHz processor and an Nvidia 7800GTX graphics card, preserving accuracy of detection in relation to the manually selected faults. It is also important to be underlined that they used a set of 2D textures to represent a 3D volume on the GPU and 32 bit floating point textures as intermediate buffers to prevent precision errors when solving the nonlinear PDE system for the diffusion, obviously because rendering to a 3D texture was not yet supported by many graphics cards.

Hee-Seok Kim et al [14] present the design and evaluation of a scalable tridiagonal solver targeted for GPU architectures. They proposed a hybrid method of tiled parallel cyclic reduction (tiled PCR) and thread-level parallel Thomas algorithm (p-Thomas). Algorithm transition from tiled PCR to p-Thomas is determined by input system size and hardware capability in order to achieve optimal performance. The proposed method is scalable as it can cope with various input system sizes by properly adjusting algorithm transition point. The tiled PCR is proposed as a variant of incomplete PCR. Specifically with tiled PCR algorithm, a large system can be divided into multiple systems and instead of loading the whole system, a large system is loaded chunk by chunk into a tile which is allocated in shared memory. In continuous the p-Thomas algorithm solves these multiple independent systems. It is also used a technique called buffered sliding window for using shared memory to process a tile efficiently. In particular, the contribution of tiled PCR was that it can minimize

global memory access, hide memory access latency due to independent tiling of a large system, and provide desirable memory layout. They measured their implementation with various combinations of different input sizes and number of input systems. The proposed method on a NVidia GTX480 showed up to 8.3x and 49x speedups over multithreaded and sequential MKL implementations on a 3.33GHz Intel i7 975 in double precision, respectively.

The cardiac monodomain model comprises a nonlinear system of partial differential equations and its numerical solution represents a very intensive computational task due to the required fine spatial and temporal resolution. Oliveira et al [15] in their paper compared three different implementations – CUDA, OpenCL and OpenGL, to a CPU multicore implementation that uses OpenMP. The PDE they solved is a reaction - diffusion equation known as the monodomain equation. In practice in every time step there are two different problems to be solved. The first one is a nonlinear system of ODEs and the second is a parabolic linear PDE. Regarding the PDE, they used the finite element method for the spatial discretization and then the second order Crank-Nikolson method. For the system of ODEs they used the explicit Euler method. In this work only 2D regular square meshes were considered, resulting in diagonally structured matrices with 9 diagonals. The compressed sparse row (CSR) format and the diagonal (DIA) format were used to store the matrices. For the OpenGL implementation the machine of rendering graphics was used to perform general purpose computations. They used the fragment processor and written the fragment shader exploiting the textures. Details for the CUDA implementation are not given. For the OpenCL implementation it is mentioned only that the CUDA solver was converted to OpenCL. The GPU implementation of the time stepping of the ODEs uses one thread to solve one ODE system, that is, each thread is associated with each node. Solving the parabolic problem includes sparse matrix - vector multiplication and solving a linear system. They used double precision arithmetic and compared the accuracy of the results using relative root-mean square. Simulations were carried out on a quad-core Intel Core i7 860 2.80GHz, 8GB of memory equipped with:(i) NVIDIA GeForce GTX 470 (ii) AMD Radeon 6850. Three different in-silico

tissue preparations were used in this work for the performance tests. For ODEs the OpenGL implementation performed better than the other two, with or without optimizations. But on the other hand CUDA implementation was the most effective between the GPU solvers. Significant large speedups were observed from all three implementations in relation to CPU OpenMP implementation for the ODE part. In the parabolic problem the computational gain was significant lower and CUDA implementation outperformed the other two. Although it is also highlighted that the OpenGL code is difficult to be understood because of its complexity while OpenCL code can be run on different accelerator devices.

Kruger and Westermann [16] with their work implemented direct solvers for sparse matrices and applied these solvers to multi-dimensional finite difference equations, i.e. the 2D wave equation and the incompressible Navier-Stokes equations. In their paper, they describe the internal representation of matrices on graphics hardware. Each vector is converted into a square 2D texture by the application program. Vectors are padded with zero entries if they do not entirely fill the texture. Moreover the transpose of a matrix is generated by simply ordering the diagonals in a different way. Off-diagonals numbered i , which start at the i -th entry of the first row, now become off-diagonals numbered $N-i$. Entries located in the former upper right part of the matrix are swapped with those entries located in the lower left part. Subsequently, they present basic algebraic operations on vectors and matrices based on this representation, such as vector arithmetic, matrix-vector product and vector reduce. The difference between sparse and full matrices just is demonstrated in that they rendered every diagonal or column vector as a set of vertices instead of set of 2D textures. The conjugate gradient (CG) method and a Gauss-Seidel solver were also implemented. With these methods equations after discretization were solved. For all these implementations they used a PC running under Windows XP on a P4 2.8GHz processor equipped with an ATI 9800 graphics card. On vectors and full matrices the implementation of standard arithmetic operations, was about 12-15 times faster compared to an optimized software implementation on the same target architecture. A considerable speed-up was achieved by internally storing vectors and matrices as RGBA

textures. On average, the multiplication of two vectors of length 5122 took 0.2 ms. Performance dropped down to 0.72 ms and 2.8 ms for vectors of length 10242 and 20482, respectively. Multiplication of a 40962 full matrix times a vector was carried out in roughly 0.23 seconds. In contrast, the multiplication of a sparse banded matrix of the same size, which was composed of 10 non-zero diagonals, took 0.72 ms. They faced problems with numerical accuracy but their performance was better compared to software implementations of BLAS library.

Daisuke Sato et al [17] use GPUs to accelerate two simulations models of electrical wave propagation in cardiac tissue, an anatomic rabbit ventricular model with 'fiber rotation' and the 2D homogeneous sheet. Each model was simulated using both the GPUs and CPUs using single precision. The cardiac tissue was modeled using the following PDE: $\frac{\partial V}{\partial t} = -1/C_m + \nabla * D \nabla V$ where V is the transmembrane voltage, I is the total ionic current, C_m is the transmembrane capacitance, and D is the diffusion tensor. This reaction diffusion equation solved with the forward Euler method, using the technique of operator splitting. For each time step, the ODE part was solved once and the PDE part was solved four times for the 2D simulation and six times for the 3D simulation. The methodology followed was to split the program into three parts, the ODE calculation, the PDE calculation, and the data transfer. They measured separately the time elapsed to data transfer and then the time for PDE and ODE calculation. The GPU simulation was performed with a single NVIDIA Geforce 8800 GT and an NVIDIA Geforce 9800 GX2 with CUDA version 1.1. The CPU simulation was performed with an 8-node high performance-computing cluster. Each node has two dual-core 2.0 GHz AMD Opteron processors. For the cluster MPI 1.0 was used. The computational speed of 2D tissue simulations with a single GPU was about 30 times faster than with a single Opteron processor. For the 3D model the computational speed with a single GPU was 1.6 times slower than with a 32- CPU Opteron cluster. Moreover GPUs cluster presented 2x speedup compared to CPUs cluster. In conclusion, it is important to be referred that the main bottleneck of the computation is the PDE and ODE part for GPU and CPU accordingly.

Mena and Rodriguez [18] present results obtained using HESIC, a novel electro-

physiology simulation software entirely developed in CUDA. The multi-scale nature of the electrophysiology problem makes difficult its numerical solution because of the millions degrees of freedom in models. These models have to be solved for thousand time steps. The use of algorithms with higher level of parallelism in multi-core platforms could be a solution. So they implemented implicit and explicit solvers in CUDA for the monodomain model using operator splitting and the finite element method. There are two parts of the parallel implementation. The one concern the solution of the system of ODEs at each mesh point and the second part is the solution of the linear system of equations associated with the parabolic PDE. All data is stored using sparse matrix structures. For the PDE solver in GPU they used CUSP and Thrust libraries, while matrices are stored in compressed sparse row sparse format and then transformed to an efficient sparse matrix format when transferred in the GPU memory for computations. Performance results are compared with an explicit multi-CPU based software. GPU simulations were run on a computer node with two Intel-Xeon Quad-Core CPUs. The node is equipped with four Nvidia Tesla M2090 GPUs. All simulations were run in a single GPU. A single GPU thread is about 478 slower than a single CPU core. However, as they underlined, theoretically the speedup could reach the range of 180x for a model with more than 1 million nodes. Finally, compared to related works, they conclude that their implementation guarantees both, the stability of the ionic model and the stability of the PDE.

Li-Wen Chang et al [19] present a solver based on the SPIKE algorithm for partitioning a large matrix into small independent matrices which can be solved in parallel using a general 1-by-1 or 2-by-2 diagonal pivoting algorithm. There are two contributions of this work. The first is that the proposed solver is the first numerically stable tridiagonal solver for GPUs and also scalable to multiple GPUs and CPUs. The second is the presentation of two optimizations: a high throughput data layout transformation for memory efficiency and a dynamic tiling approach for reducing the memory access footprint caused by branch divergence. The solver begins by creating the partitions of the input matrix and vector, preparing for the parallel solver step. The parallel partitioned solver for the collection of independent systems then

computes both the solutions to the partitions of the original system as well as the components of the spike matrix S . Then a solver for the spike system is invoked, first solving the reduced spike system, followed by the computation of all remaining values in the solution X through backward substitution. Finally, the results from all the different partitions are collected and merged into the expected output format and returned. In the parallel partitioned system solver step, any solver can be used to solve the independent systems. The authors in their experiments provided both their thread-parallel diagonal pivoting solver and a parallel Thomas solver. For the data layout transformation the strategy was in the partitioning step of the SPIKE algorithm, to marshal the data such that Element K of Partition I is adjacent to Element K from partitions $I-1$ and $I+1$. This strategy can guarantee better memory efficiency in the kernel execution but pay the cost of data marshaling. For the data tiling the idea followed was to use a dynamic tiling mechanism, which bounds the size of the access footprint from the threads in a warp. The original while loop is dynamically tiled to a few smaller while loops. Barrier synchronization is put between the smaller while loops to force "fast" threads to wait for "slow" threads. The evaluation tests performed in 16 types of nonsingular tridiagonal matrices of size 512. The machines used are a single node of the NCSA GPU Accelerated Cluster (AC) with an Intel Xeon X5680 CPU and 2 NVIDIA GTX480 GPUs and a NCSA GPU Forge cluster with 2 AMD Opteron 6128 CPUs and 6 or 8 NVIDIA M2070 GPUs. The evaluation of the performance of the GPU-based solver was compared against the results of CUSPARSE library on a GPU and MKL library on a CPU, on the AC machine. In the case of a random matrix, the proposed method performs with comparable execution time (less than 5% difference) to that of CUSPARSE, while in terms of precision; the authors' SPIKE diagonal pivoting has a much smaller error rate. Compared to MKL, their method can get 3.20x and 12.61x speedups with and without considering data transfer from GPU, respectively. The SPIKE-Thomas implementation, which is numerically stable in many cases but cannot be guaranteed, outperforms CUSPARSE by a factor of 1.72x, and compared to MKL, it shows 3.60x and 22.70x speedups with and without data transfer, respectively. For a strictly column diagonally dom-

inant matrix both CUSPARSE and the SPIKE-Thomas implementation are stable. The SPIKE-diagonal pivoting shows 1.35x, 3.13x, 16.06x speedups over CUSPARSE, MKL with and without considering data transfer for GPU, respectively. The scalability evaluation performed in NCSA Forge on 1,2,4,8 and 16 GPUs. The results of 1, 2, and 4 GPUs use on single node, while the results of 8 and 16 GPUs are generated on multiple nodes. In the first case a strong but not linear scaling was observed. In the second case, the GPU-based solver achieved perfectly linear scalability when the input begins already distributed among the various nodes. Finally, the library does scale well when the matrix size increases.

Gaikwad and Toke [20] presented GPU based parallel implementations of Krylov subspace based iterative solvers for solving several small sized systems arising from this method. They selected Stabilized BiConjugate Gradient (BiCGStab) and Conjugate Gradient Squared (CGS) methods for the solutions of sparse linear systems with unsymmetric coefficient matrices. They considered the phenomenon of the curse of dimensionality so as to use sparse grid combination technique. In this solver the goal was to solve problems with irregular sparsity. Their solver implementations used CUBLAS Library mainly for vector-vector operations. CUBLAS does not provide sparse matrix storage structures, so they considered general storage formats and which of them could be more suitable. They experimented with three libraries (NVIDIA SpMV Library, IBM SpMV Library and CNC SpMV Library) to implement the linear solvers. For evaluating the GPU implementations of linear solvers they chose Black-Scholes partial differential equation. The sparse grid combination technique allowed them to solve for higher dimension by reducing the size of the problem. All sub-grids were solved sequentially a single GPU using BiCGStab and CGS solvers. These solvers on CPU developed for double precision, while on GPU, due to limitations of Tesla C870, the solvers were done with single precision arithmetic. In their paper it was referred analytically, using tables, the performance for each experiment. Experiments were performed on NVIDIA Tesla C870 and Intel Xeon E5420. For small grids, both CPU and GPU implementations exhibit equivalent performance but GPU solvers achieve better speedups for the grids with large unknowns than for

the smaller grids. Their results showed that the choice of sparse format is not only important for scalability of iterative solvers to solve the sparse grids but the efficient implementation and parameter tuning of matrix-vector kernels is also essential for maximal performance on GPUs.

Glimberg et al [21] implemented the GPULab Library which is a GPU-based Framework for PDE Solvers created by the Section of Scientific Computing in the Department of Informatics and Mathematical Modeling of Technical University of Denmark. They have decided to invest time now to develop a generic framework, in order to easily solve a broad range of PDE problems in the future (Inspired by the PETSc framework). Their key components for High performance PDE-solvers are stencil based flexible order FD operations, iterative methods for solving large systems of equations (mixed precision) and decomposition techniques. Currently, they are working with the OceanWave3D model for coastal and offshore engineering. In the future they want to solve large problems fast because currently they are limited by the GPU memory. They want to be limited by the total number of GPUs. They intent to solve on multiple GPUs, on multiple workstations.

Jeff Bolz et al [22] implemented two basic computation kernels on GPUs, a sparse matrix conjugate gradient (CG) solver for unstructured grids using a mesh with 37k vertices and a regular grid multigrid solver on a fluid-flow problem which requires a Poisson solver with Neumann boundary conditions. Implementing a CG solver requires a sparse matrix-vector multiply and vector inner-product. Each of the matrix A and vector x are stored in textures requiring appropriate indirections. The sparse matrix A is stored in two textures, one for the diagonal entries and one for the off-diagonal, non-zero entries of A . They have implemented all of the components of a general conjugate gradient solver, as well as the specific matrices for geometric flow, including their recomputation for each smoothing step. Their optimizations focused on the rectangles dimensions and layout. The tests were performed in NVIDIA's GeForce FX GPU's. They implemented CPU versions of the matrix multiply kernels using SSE, and tested them on a 3GHz Pentium 4. The GPU implementation achieved 120 unstructured matrix multiplies per second whereas the CPU implementation can

only do 75 per second on the stated problem instance. For the multigrid solver for discretization of elliptic PDEs over regular grids was considered the Helmholtz equation with Dirichlet and/or Neumann boundary conditions on the unit square. After discretization remains to be solved a linear system. For the structured matrix multiply, the GPU can do 1370 matrix multiplies per second whereas the CPU can do 750 per second. The tests have shown that both the CPU and GPU implementations are bandwidth limited and that the multigrid solver has enormous performance potential, and would be even more useful if it were applied to irregular grids.

Yao Zhang et al [24] study the performance of three parallel algorithms and their hybrid variants for solving tridiagonal linear systems on a GPU: cyclic reduction (CR), parallel cyclic reduction (PCR) and recursive doubling (RD). They found that CR enjoys linear algorithm complexity but suffers from more algorithmic steps and bank conflicts, while PCR and RD have fewer algorithmic steps but do more work each step. To combine the benefits of the basic algorithms, they propose hybrid CR+PCR and CR+RD algorithms. They decided to develop hybrid methods due to some observations. First all three algorithms have fine-grained parallel structures which are suitable for GPU programming. With respect to computational complexity, CR is the best algorithm because it is $O(n)$, while PCR and RD are $O(n \log_2 n)$. However, CR suffers from a lack of parallelism at the end of the forward reduction phase and at the beginning of the backward substitution phase. On the other hand, although PCR and RD have fewer algorithmic steps, they always have more parallelism through all steps. The hybrid methods improve CR by switching to PCR or RD to reduce inefficient steps when there is not enough parallelism to keep a GPU busy. The hybrid algorithms first reduce the system to a certain size using the forward reduction phase of CR, then solve the reduced (intermediate) system with the PCR/RD algorithm. Finally, they substitute the solved unknowns back into the original systems using the backward substitution phase of CR. For each system, the three diagonals and right-hand side were loaded from global memory to shared memory, solve the system, and store the solution back to global memory. Therefore global memory communication only occurs at the beginning and end of all algorithms. With the used hardware, systems

of more than 512 equations would exceed the size of shared memory. The proposed solvers do support this case at a cost of roughly 3x performance degradation by using global memory only. CR has the least work (shared memory accesses and arithmetic operations) but the most steps, whereas PCR and RD have fewer steps but more work. This motivates an approach that takes advantage of the best parts of both: doing the least work when there is sufficient parallelism at least of warp size, but then switching to performing fewer steps when there is not enough parallelism to fill the machine. The switch is actually even more beneficial because there are bank conflicts in the CR solver and shared memory access dominates the execution time. Experiments performed in NVIDIA GTX 280 and Intel Core 2 Q9300. Hybrid algorithms improved the performance of PCR, RD and CR by 21%, 31% and 61% respectively. Their GPU solvers achieve up to a 28x speedup over a sequential LAPACK solver, and a 12x speedup over a multi-threaded CPU solver.

Göddeke and Strzodka [23] in their paper present a new implementation of cyclic reduction for the parallel solution of tridiagonal systems and employ this scheme as a line relaxation smoother in their GPU-based multigrid solver. They also reevaluate the mixed precision solvers, shown in their previous work, that run entirely on the GPU. Among some previous works about parallel algorithms for the solution of tridiagonal equations systems, they pointed the work by Zhang et al [24]. The last one conclude that cyclic reduction suffers from shared memory bank conflicts and poor thread utilization in lower stages of the solution process, while parallel cyclic reduction is not asymptotically optimal and recursive doubling is not optimal and additionally exhibits numerical stability issues. Zhang et al [24] developed a hybrid combination of cyclic reduction and parallel cyclic reduction to alleviate these deficiencies. On the other hand, Göddeke and Strzodka implemented a cyclic reduction algorithm that exhibits much better memory access patterns and reached the same performance as their best hybrid algorithm. As a test problem they solved the Poisson problem on isotropic and anisotropic domains with homogeneous Dirichlet boundary conditions. After discretization on rectangular domains they obtained a linear system of equations ($ALx = b$, logical tensor product structure). To measure accuracy, they evaluated the

analytical Laplacian of the polynomial function $u_0(x, y) = x(a - x)y(b - y)$ at the grid points and used the resulting coefficient vector as the right hand side of the linear system. It is shown that attempting to solve the system in single precision failed, while double precision sufficed to reduce the error according to finite element theory and to guarantee the result accuracy. Nevertheless, single precision is 2.5 times faster. So they used mixed precision solvers. As it is referred the mixed precision solver for a linear system $Ax = b$ basically comprises the following steps: First, compute $d = b - Ax$ in double precision. Second, solve $Ac = d$ approximately in single precision. Third, update $x = x + c$ in double precision. Final check for convergence and iterate. Subsequently, they analyzed some multigrid smoothers, the Jacobi, the Gauss-Seidel and finally Thomas algorithm as a tridiagonal solver for CPU and Cyclic reduction as a tridiagonal solver for GPU. The CR algorithm proceeds in two steps: a forward reduction and backward substitution. The main problem of this algorithm is the bank conflicts in the on-chip memory which reduce the internal bandwidth. Their solution to this problem is to group the indices in each level of the reduction tree in two contiguous arrays of odd and even indices. When they load the initial data into shared memory they already separate even and odd indices. With an appropriate padding between the arrays this is a bank conflict free read operation from global to shared memory. The output of each forward update step writes again into separate even and odd arrays. Moreover their implementation directly permutes the matrix bands after the assembly, and stores both row- and column oriented coefficients. Consequently, both sets of matrix bands are passed to the device. Their tests were performed on an NVIDIA GeForce GTX 280 and Intel Core2Duo E6750 CPU. The Poisson problem $-\Delta u = f$ on a unit square domain $= ([0, 1])^2$ with Dirichlet boundary conditions. The GPU features 30 multiprocessors supports double precision. They used a multigrid solver for their tests. A preconditioned conjugate gradient solver treated the coarse grid problems. Its preconditioner was either Jacobi or ADI-TRIDI. The stopping criterion for the solver was set to reduce the initial residual by eight digits. They performed some accuracy studies about mixed and double precision and they showed the results analytically. Finally, they had speed up measurements about the smoothers.

The speedup factors are smaller in the fairer comparison of the two mixed precision solvers, but still reach almost an order of magnitude, which clearly highlights the advantages of the GPU. At least a 70% speedup is achieved on the device alone by the mixed precision scheme, often reaching a factor of two. Different problems required different smoothers for optimal performance.

3.4 Conclusions of Review

Having study all this work that it was described above, we can conclude matrices more often are stored in compressed sparse row (CSR) and then transformed to an efficient sparse matrix format when transferred in the GPU memory for computations. It is remarkable that the majority of the results were compared to CUBLAS, CUSPARCE and MKL libraries. In some cases it is noticed that GPU implementations were compared to CPU parallel implementations written by the authors. About discretization, the first step of solving PDEs, it is shown in the figure follows that the most popular technique was the finite different with the finite element to be second. A large amount of research papers did not mention the technique that they followed.

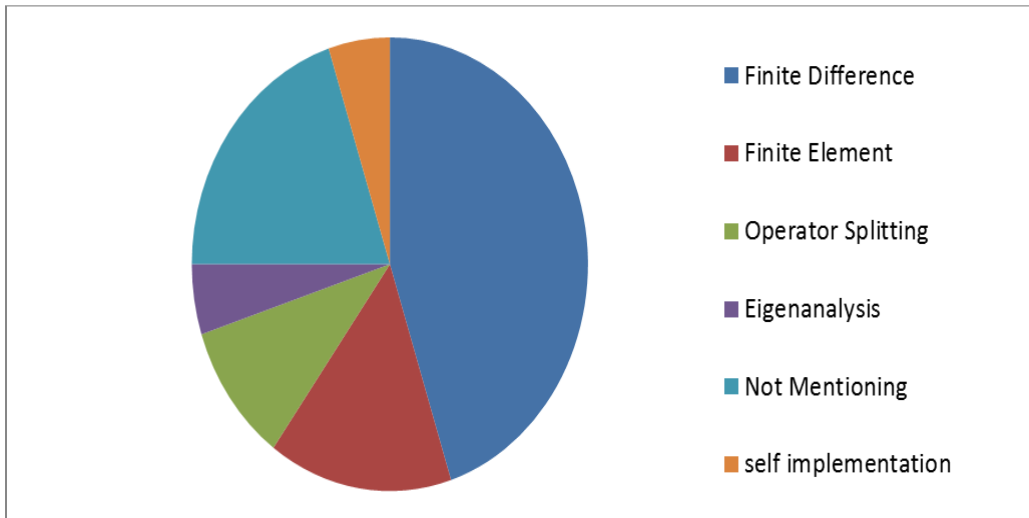


Figure 3-1: Discretization techniques

Finally, an important conclusion about all the above papers concerns the methods

that were most implemented for solving linear systems. In the following table, it is presented for each work the discretization method, the equation and the solver that was implemented. Most popular methods are Parallel Cyclic Reduction (PCR), Conjugate gradient (CG) and Jacobi.

#	Discretization Method	PDE equation	Linear System Solver
1	Finite Difference	3D heat diffusion equation	Jacobi
2	Finite Difference	2D/3D Wave equation	-
3	Finite Difference	2D parabolic PDE	-
4	-	-	-
5	"naïve algorithm", "cached plane sweep", "pipelined wavefront"	-	-
6	Finite Difference	the Cahn-Hilliard equation	-
7	Finite Difference	the parabolic convection diffusion equation	Parallel cyclic reduction
8	Finite Difference	focus on stochastic volatility models, the Hundsdorfer-Verwer scheme and the Douglas scheme	ADI schemes
9	Eigenanalysis	directional anisotropic diffusion & coherence enhancing diffusion	Explicit Euler scheme and an iterative method(self implemented)
10	Not Mentioning	-	Hybrid tiled PCR & p-Thomas
11	Finite Element	The cardiac monodomain model (non linear ODE's and parabolic linear PDE's)	2nd order Crank-Nikolson (PDE) & explicit Euler (ODE)
12	Finite Difference	the 2D wave equation & the incompressible Navier-Stokes equations	CG & Gauss-Seidel
13	operator splitting	Reaction diffusion equation	Forward Euler
14	Finite Element and operator splitting	the monodomain model	Implicit and explicit solvers
15	-	-	SPIKE algorithm ,parallel diagonal pivoting & p-Thomas
16	-	Black-Scholes partial differential equation	BiCGStab and CGS
17	Finite Difference	OceanWave3D model	Iterative methods
18	-	Helmholtz equation with Dirichlet and/or Neumann boundary conditions on the unit square	CG and Multigrid solvers
19	Finite Element	Poisson problem on isotropic and anisotropic domains	Multigrid, Jacobi, Gauss-Seidel, Thomas(CPU) and parallel Cyclic Reduction(GPU)
20	-	-	CR,PCR,RD, Hybrid CR+PCR,CR+RD
21	Finite Difference	the parabolic convection diffusion equation	Parallel cyclic reduction

Table 3.2: Conclusion Table

Chapter 4

Implementation

4.1 Cyclic Reduction

Cyclic reduction (CR) is an algorithm introduced by G. H. Golub and R. W. Hoekney [26] in the mid 1960s for solving tridiagonal linear systems related to the finite difference discretization of the Poisson equation over a rectangle. The basic idea of the CR algorithm, also called odd-even reduction, is to repeatedly reduce the system to half size until one equation is left and solve backwards to find all unknowns.

4.1.1 The algorithm

CR method only applies to matrices that can be represented as a (block) Toeplitz matrix; such problems often arise in implicit solutions for partial differential equations on a grid. For example fast solvers for Poisson's equation express the problem as solving a tridiagonal matrix, discretising the solution on a regular grid. From 1D Poisson's equation arises a tridiagonal matrix system while from 2D Poisson's equation arises a block tridiagonal system. In general large tridiagonal systems of linear equations appear in many numerical analysis applications. A tri-diagonal matrix is a matrix with values only in the sub-, main- and super-diagonal. For example, the tridiagonal system:

$$a_i * x_{i-1} + b_i * x_i + c_i * x_{i+1} = F_i, \quad i = 1 : 1 : n$$

Assume that $n = 2^p - 1$. The algorithm proceeds in two steps: a forward reduction and a backward substitution. Each phase consists of $(\log_2 n - 1)$ steps, where n is the system size. The first step in cyclic reduction is to combine linearly the equations in order to eliminate the odd numbered unknowns $(x_1, x_3 \dots x_n)$. Next the unknowns are re-ordered and the process is continued until the system consists from one equation with one unknown. To do this the algorithm is based to triplets. In the above example in order to eliminate x_1 and x_3 , the first three equations of the system are chosen and multiplied by the parameters $\alpha_2, \beta_2, \gamma_2$.

$$\begin{array}{ll}
 b_1 * x_1 + c_1 * x_2 = F_1 & *(\alpha_2) \\
 a_2 * x_1 + b_2 * x_2 + c_2 * x_3 = F_2 & *(\beta_2) \\
 a_3 * x_2 + b_3 * x_3 + c_3 * x_4 = F_3 & *(\gamma_2)
 \end{array}$$

Subsequently the equations are summed and the resulted equation is produced. Similarly, this elimination process proceeds for the next three equations, until only one equation left. Using the back substitution the unknown x can be calculated from the last one equation and all the others x_i can be found from the previous steps . The equations involved in all these stages are:

$$\begin{aligned}
 a'_i &= -a_{i-1} * k_1 \\
 b'_i &= b_i - c_{i-1} * k_1 - a_{i+1} * k_2 \\
 c'_i &= -c_{i+1} * k_2 \\
 d'_i &= d_i - d_{i-1} * k_1 - d_{i+1} * k_2 \\
 \text{where } k_1 &= \frac{a_i}{b_{i-1}}, k_2 = \frac{c_i}{b_{i+1}} \\
 x_i &= \frac{d'_i - a'_i * x_{i-1} - c'_i * x_{i+1}}{b'_i}
 \end{aligned}$$

The described steps can be illustrated in Figure 4-1.

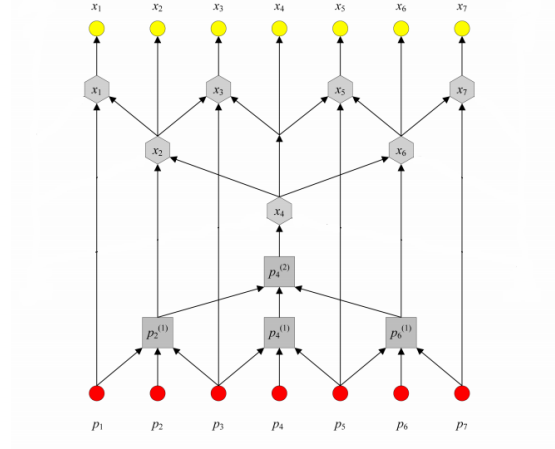


Figure 4-1: CR Method

4.1.2 Implementation Issues

As it was said a tridiagonal matrix has basically 3 diagonals, (super, main, sub). To take advantage of this, only these three diagonals vectors plus one vector for the right hand side of the system and one vector for the solution have to be stored. Vectors a , b , c are used to hold the sub-diagonal, the main diagonal and the super-diagonal respectively. Vector F holds the right hand side and U the solution vector. The code implemented for all the steps of the algorithm is given below:

Algorithm 1 Cyclic Reduction - Forward

```
1: function FORWARD REDUCTION
2:   for( $i = 0; i < \log_2(size + 1) - 1; i++$ ) {
3:     for( $j = 2^{i+1} - 1; j < size; j = j + 2^{i+1}$ ) {
4:        $offset = pow(2, i);$ 
5:        $index_1 = j - offset;$ 
6:        $k_1 = a[j]/b[index_1];$ 
7:        $k_2 = c[j]/b[j];$ 
8:        $b[j] = b[j] - c[j - offset] * k_1 - a[j + offset] * k_2;$ 
9:        $F[j] = F[j] - F[j - offset] * k_1 - F[j + offset] * k_2;$ 
10:       $a[j] = -a[j - offset] * k_1;$ 
11:       $c[j] = -c[j + offset] * k_2;$ 
12:    }
13:  }
14: end function
```

After the forward elimination the resulting system is one equation with one unknown. Hereafter, it is trivial to deduce the middle unknown of the system.

Algorithm 2 Solve the middle equation

```
1: function FIND MIDDLE
2:    $intindex = (size - 1)/2;$ 
3:    $x[index] = F[index]/b[index];$ 
4: end function
```

Afterwards, the algorithm backwards to calculate repeatedly the remaining unknowns. Substantially, the resulting solution is calculated by the following code:

Algorithm 3 Cyclic Reduction - Backward

```
1: function BACKWARD SUBSTITUTION
2:   for( $i = \log_2(size + 1) - 2; i \geq 0; i--$ ) {
3:     for( $j = 2^{i+1} - 1; j < size; j = j + 2^{i+1}$ ) {
4:        $offset = 2^i$ ;
5:        $index_1 = j - offset$ ;
6:        $index_2 = j + offset$ ;
7:       if( $j! = index_1$ )
8:          $x[index_1] = (F[index_1] - a[index_1] * x[index_1 - offset] -$ 
           $c[index_1] * x[index_1 + offset]) / b[index_1]$ ;
9:       if( $j! = index_2$ )
10:         $x[index_2] = (F[index_2] - a[index_2] * x[index_2 - offset] -$ 
           $c[index_2] * x[index_2 + offset]) / b[index_2]$ ;
11:     }
12:   }
13: end function
```

The first attempt was to simply port the above algorithm on GPU. Initially three kernels were obtained based on the three independent steps (Forward Reduction, Find middle, Backward Substitution) shown above. At this point, data dependencies were observed between iterations of the external loop both in Forward Reduction kernel and in Backward Substitution kernel. Subsequently, the first Kernel for forward elimination called in this way:

```
for (i=0; i<log2(size+1)-1; i++){
  <Kernel forward launch>
}
```

Variable i holds the iteration of the loop and passes as parameter in the kernel, which essentially settles on the termination condition. In a symmetric way the Backward Substitution kernel is launched, but with the iterations in decreasing order. The kernel Find middle is launched by one thread only, since it is actually one computa-

tion. One essential optimization that was performed and increased significantly the speed up, as is will be shown in chapter five, is the dynamic calculation of the block dimension. It was observed that according to the iterations of the external loop, the number of the threads actually doing a job, was affected. So it was considered the following function in order to modify the size of the block dynamically:

Algorithm 4 Block Dimension

```

1: function CALCULATE BLOCK DIMENSION(size,block,grid)
2:   if( $size < 4$ ) { $block \rightarrow x = 1; block \rightarrow y = 1;$ }
3:   else if( $size < 16$ ) { $block \rightarrow x = 2; block \rightarrow y = 2;$ }
4:   else if( $size < 64$ ) { $block \rightarrow x = 4; block \rightarrow y = 4;$ }
5:   else if( $size < 256$ ) { $block \rightarrow x = 8; block \rightarrow y = 8;$ }
6:   else { $block \rightarrow x = 16; block \rightarrow y = 16;$ }
7: end function

```

In other words, the geometry changes while the size of the system is growing. Likeness for the backward substitution step, the block dimension is determined.

Another optimization considered was padding. Padding is a technique often used to avoid divergence between threads in a warp, caused mainly by branches. This technique is implemented by increasing the allocated memory in order to eliminate the branches. In this implementation, padding was used to remove the if - branches in Backward Substitution kernel. These if - branches were present to preserve that threads would not exceed the allocated memory size. The maximum offset that threads could exceed was $\log_2(size + 1)$. So the padded vectors was $2 * \log_2(size + 1)$ bigger in order to cover both directions. Performance results are demonstrated in Chapter 5.

In relation to memory transfers, before the kernels launches, all vectors except vector U are transferred from host to device using continuous CudaMemcpy(). It was preferred to initialize to zero the resulting vector U using CudaMemSet() instead of copying it through CudaMemcpy() due to lower cost. Finally, the only vector which is necessary to be copied back from the device to host is the solution vector U.

4.2 Block Cyclic Reduction

The basic idea of the cyclic reduction method can be extended to block tridiagonal systems. The idea of the block cyclic reduction (BCR) was first introduced by Gene Golub to deal with the scalar tridiagonal systems that arise from the finite element discretization of the Poisson equation in 2D. As in cyclic reduction, block cyclic reduction is a two phase algorithm. It consists of forward reduction and backward substitution. During each step of the reduction stage, are eliminated approximately half the unknowns in the system. After $O(\log_2(n+1))$ reductions a 1×1 block system is left. After solving this system, the previously eliminated unknowns are computed by back substitution. While this formulation is numerically unstable, O.Buneman suggested a stable variation[]. In this thesis, we consider the case of block cyclic reduction, where the scalar elements of traditional cyclic reduction are replaced with matrix tridiagonal and diagonal blocks.

4.2.1 The algorithm

The implemented algorithm solves systems of the following form: $A * U = F$ where the A is a block tridiagonal matrix:

$$\begin{pmatrix} B & T & & & \\ T & B & T & & \\ & T & B & T & \\ & & \dots & & \\ & & & \dots & \\ & & & T & B & T \\ & & & & B & T \end{pmatrix}$$

and B is a tridiagonal matrix. For example a typical tridiagonal matrix arising from Poisson discretization is :

the *B Matrix*

$$\begin{pmatrix} -4 & 1 & \\ 1 & -4 & -1 \\ & -1 & -4 \end{pmatrix}$$

and T is either the *Identical Matrix*

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

or The *-Identical Matrix*

$$\begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

The concept of block cyclic reduction is to iteratively eliminate half of the unknowns until there is an only single block system which can be solved directly. So we have for such as : $1 < j < 2^{jq} - 1$

$$TU_{2*j-2} + BU_{2*j-1} + TU_{2*j} = F_{2*j-1}$$

$$TU_{2*j-1} + BU_{2*j} + TU_{2*j+1} = F_{2*j}$$

$$TU_{2*j} + BU_{2*j+1} + TU_{2*j+2} = F_{2*j+1}$$

By multiplying the first and third equations by T and the second equation by $-B$, and add the three equations, if $TB = BT$ the odd unknowns U_{2*j-1} are eliminated:

$$T^2U_{2*j-2} + (2T^2 - B^2)U_{2*j} + T^2U_{2*j+2} = TF_{2*j-1} - BF_{2*j} + TF_{2*j+1}$$

Then the same structure occurs for this new linear system with half of the unknowns. If this procedure continues for k steps until $k = jq - 1$ then remains only

one block equation in the system :

$$B^{(jq-1)}U_{2^{jq-1}} = F_{2^{jq-1}}^{(jq-1)}$$

After solving the one block equation, by solving the linear system, the backward substitution phase begins. So after the even values were computed in the previous step , the "odd" values are now calculated.

4.2.2 Implementation Issues

Block Cyclic Reduction is not often implemented due to high storage demands of the algorithm. In particular instead of elements, block of elements have to be stored. In every calculation step of the forward reduction, matrices B and T are modified and need to be stored in order to be used in the backward phase. There were two options to deal with this demand. The one was to store all calculated matrices and the other to re-compute them every time. In view of the fact that GPU is more efficient handling large amount of computation and suffers from limited memory size the second option was chosen.

The inverse of table B in every step it is computed as a function of the original table B, an angle θ and a number α so as to avoid storing all the middle results and changes on table B. Bellow is presented the formula for this computation where Chebychev polynomials are used:

$$\begin{aligned} T^{(k)} &= T^{2^k} \\ B^{(k)} &= - \prod_{l=1}^{2^k} (B - 2 * \cos(\theta_{kl}) * T) \\ [B^{(-1)}]^{(k)} &= - \sum_{l=1}^{2^k} \alpha_{(k*l)} * [B - 2 * \cos(\theta_{k*l}) * T]^{(-1)} \\ \theta_{k*l} &= (l - 1/2) * \pi / 2^{jq} \\ \alpha_{k*l} &= (-1)^l / 2^k * \sin(\theta_{k*l}) \end{aligned}$$

The pseudocode for the Buneman algorithm that was implemented goes as following:

Algorithm 5 Block Cyclic Reduction - Buneman

```
1: function BUNEMAN
2:    $q = \text{blocksize}, n = \text{size}, jq = \log_2(n + 1);$ 
3:    $P = \text{zeros}(n, q, jq), Q = \text{zeros}(n, q, jq);$ 
4:    $U = \text{zeros}(n, q), F = \text{ones}(n, q);$ 
5:   For( $j = 1 : 1 : 2^{jq} - 1$ )
6:      $Q(j, :)^{(0)} = F(j, :)^{(0)};$ 
7:      $P(j, :)^{(0)} = 0;$ 
8:   End For
9:   For( $k = 1 : 1 : jq - 1$ )
10:     For( $j = 1 : 1 : 2^{jq-k} - 1$ )
11:        $idx_1 = 2^k * j;$ 
12:        $idx_2 = 2^{k-1};$ 
13:        $P_{idx_1}^{(k+1)} = P_{idx_1}^{(k)} * (T^{(k-1)} * P_{idx_1-idx_2}^{(k)} + P_{idx_1+idx_2}^{(k)} - Q_{idx_1}^{(k)});$ 
14:        $Q_{idx_1}^{(k+1)} = T^{(k-1)} * (Q_{idx_1-idx_2}^{(k)} + Q_{idx_1+idx_2}^{(k)} - 2 * T^{(k-1)} * P_{idx_1}^{(k+1)});$ 
15:     End For
16:   End For
17:    $pointer = 2^{jq-2(jq-1)};$ 
18:    $U_{pointer} = [B^{-1}]^{(jq-1)} * Q_{pointer}^{(jq)} + Q_{pointer}^{(jq)};$ 
19:   For( $k = jq - 1 : -1 : 1$ )
20:     For( $j = 2 : 1 : 2^{jq-k} - 1$ )
21:        $pointer = 2^k * j - 2^{(k-1)};$ 
22:        $U_{pointer} = [B^{-1}]^{(k-1)} * (Q_{pointer}^{(k)} - T^{(k-1)} * (U_{2^k*j} + U_{2^k*j-2^k})) + P_{pointer}^{(k)};$ 
23:     End For
24:   End For
25:    $j = 1$ 
26:    $pointer = 2^k - 2^{(k-1)};$ 
27:    $U_{pointer} = [B^{-1}]^{(k-1)} * (Q_{pointer}^{(k)} - T^{(k-1)} * U_{2^k}) + P_{pointer}^{(k)};$ 
28:    $j = 2^{jq-k};$ 
29:    $pointer = 2^{jq} - 2^{(k-1)};$ 
30:    $U_{pointer} = [B^{-1}]^{(k-1)} * (Q_{pointer}^{(k)} - T^{(k-1)} * U_{2^{jq}-2^k}) + P_{pointer}^{(k)};$ 
31: end function
```

For the serial version of the algorithm, Intel’s Math Kernel Library (MKL) [] was used. MKL is a library of optimized math routines for science, engineering, and financial applications. Core math functions include BLAS, LAPACK, ScaLAPACK, sparse solvers, fast Fourier transforms, and vector math. For the serial code running on CPU, we used LAPACK and BLAS routines for many math operations. Specifically, the routines which was used was for the following BLAS operations:

- Matrix-Matrix multiplication
- Matrix-Vector Multiplication
- Inverse Matrix
- Matrix addition
- Vector addition
- Scalar Matrix Multiplication

Accordingly for the code running on GPU we used the optimized routines from cuBLAS an optimized BLAS library for NVIDIA based GPU cards and CULA an optimized library that implements LAPACK routines for GPUs.

The first optimization performed was to restructure the code by merging math operations. In this way, routines of the above-mentioned libraries used optimally, taking advantage of the kernels’ characteristic to increase performance as the amount of data increases.

Another attempt to raise the efficiency of the algorithm was to avoid the inversion of the matrix B, an in general very expensive operation. One way to achieve this is the replacement of the inversion with a linear system solution. Particularly, in the above pseudo-code in lines 18,22,27 and 30 where the calculation of the inverse of matrix B occurs applied the below strategy:

$$B^{-1} * b = x \implies B * x = b$$

For example in line 22 all the expression: $(Q_{pointer}^{(k)} - T^{(k-1)} * (U_{2^k*j} + U_{2^k*j-2^k}))$ was considered as a b vector. This vector constituted the right hand side of the system $B*x = b$ where B the matrix defined above. By solving this system, a solution vector x was computed. This vector x substituted all the expression:

$$[B^{-1}]^{(k-1)} * (Q_{pointer}^{(k)} - T^{(k-1)} * (U_{2^k*j} + U_{2^k*j-2^k}))$$

This method was applied in every appearance of B^{-1} . The last optimization came of the need to examine larger problems. Storage of Q and P matrices, that consist the Buneman Series, are the most spatial expensive. A "sliding window" technique was used. Instead of transferring all the matrices P and Q on GPU, it is preferable to load in each step two matrices to hold the current and the previous iteration. Although a lot of space is now saved and larger problems can be executed, additional memory transfers between host and device must be done.

As mentioned above the second version of Buneman implementation requires memory transfers of the P and Q matrices between the host and the device. As an extra optimization it is considered to use asynchronous copies for the memory transfers in order to overlap the transfers with computations. Finally it was remarked that the transfer cost of was significantly small to be overlapped and to contribute to the improvement of speed-up.

Performance impact of all referred optimizations is demonstrated in Chapter 5.

Chapter 5

Results

5.1 Hardware and Metrics

The hardware used for the experiments is :

- Intel Xeon CPU W3550 @3.07 GHz with 8GB RAM and four cores
- NVIDIA GeForce GTX680 (which was analytically discussed in Chapter 2)

Time was measured using `clock()` function and is presented in seconds. All experiments were performed with full optimizations turned on. CPU code was compiled with Intel's compiler (`icc`) and GPU code with `nvcc`.

5.2 Experiments

The performance results for each implementation, described in Chapter 4, are presented in the following tables . The execution time is the average of ten measurements for each case. Furthermore, the speed - up between CPU and GPU is given.

The correctness of results was validated with relative MATLAB functions. The results of the CPU have been identified with the results of the GPU.

5.2.1 Single Precision Performance

Table 5.1: Cyclic Reduction (first version) execution time (SP)

Size	CPU	GPU
2^{21}	0.113	0.02
2^{22}	0.216	0.034
2^{23}	0.455	0.063
2^{24}	0.909	0.126
2^{25}	1.82	0.25
2^{26}	3.65	0.50

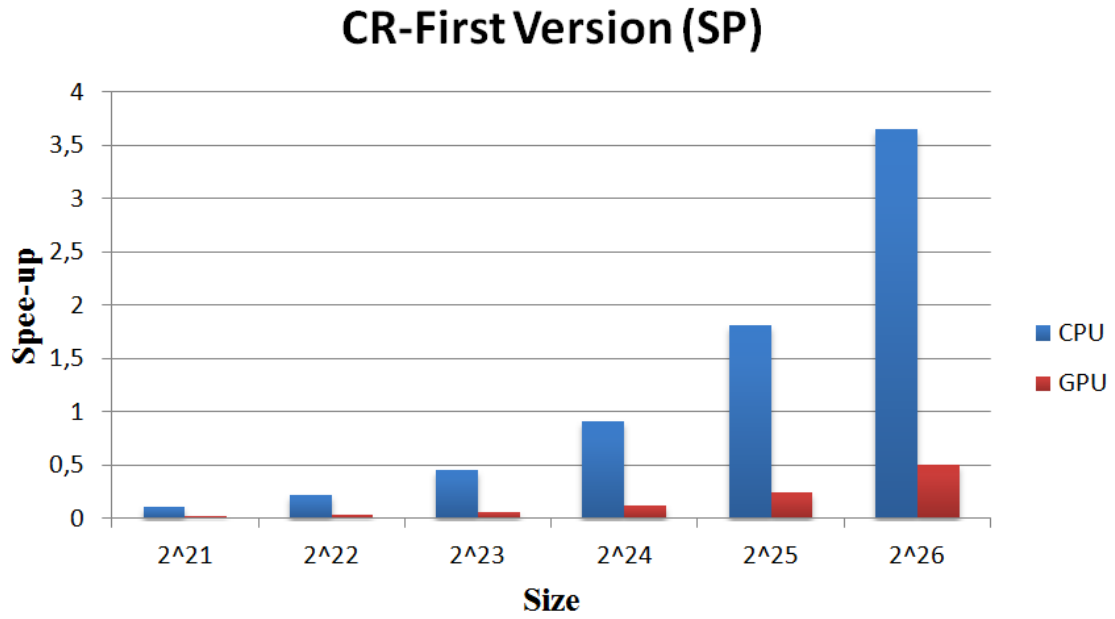


Figure 5-1: CPU-GPU Cyclic Reduction(SP)

Table 5.2: Cyclic Reduction (first version) speed - up (SP)

size	Speed - up
2^{21}	5.38
2^{22}	6.35
2^{23}	7.22
2^{24}	7.21
2^{25}	7.28
2^{26}	7.1

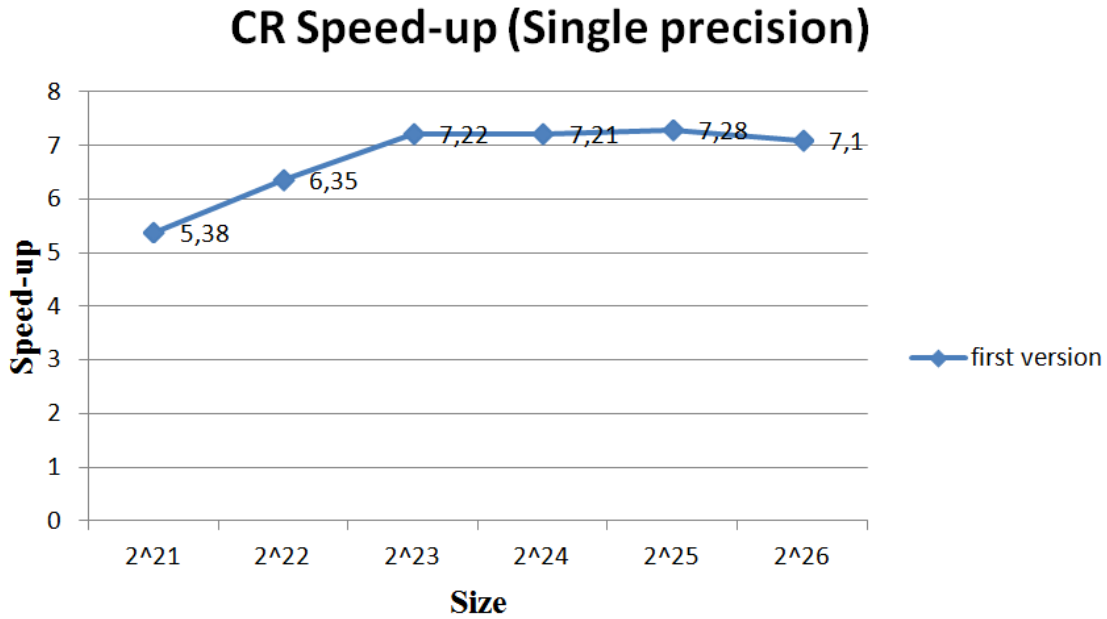


Figure 5-2: Speedup Cyclic Reduction(SP)

As can be observed from the above table the speed-up increases as the data set increases. This is normal because GPU performs better when has more data to process. It is worth mentioning that a decrease is observed from size = 2^{25} to size = 2^{26} . To be explained that measurements were made in relation to computation to memory transfers ratio. It was observed that from a certain size memory transferring consumes more time than the actual computation.

Table 5.3: Cyclic Reduction (padding version) execution time (SP)

Size	CPU	GPU
2^{21}	0.108	0.021
2^{22}	0.214	0.035
2^{23}	0.456	0.066
2^{24}	0.877	0.115
2^{25}	1.769	0.251
2^{26}	3.546	0.509

Table 5.4: Cyclic Reduction (padding version) speed - up (SP)

size	Speed - up
2^{21}	4.14
2^{22}	6.11
2^{23}	6.90
2^{24}	7.62
2^{25}	7.04
2^{26}	6.96

The padding technique did not increase the performance of the implementation. That happened because there was no significant divergence in the code.

In the following tables it is presented the speed up of the Buneman algorithm for different geometries, where Q is the size of block and N the number of blocks. In the first six tables, the size of block remains stable while the number of blocks is increasing. As GPU(first) is mentioned the first implementation and as GPU(second) the one with the sliding window optimization. GPU results were compared with sequential CPU and multithreaded (4 threads) CPU.

Table 5.5: Block Cyclic Reduction N=3

N	Q	CPU(parallel)	CPU(seq)	GPU(first)	GPU(second)
3	127	0.000	0.000	0.005	0.004
3	255	0.005	0.000	0.020	0.020
3	511	0.027	0.060	0.062	0.062
3	1023	0.168	0.450	0.145	0.147
3	2047	1.058	3.160	0.403	0.403
3	4095	7.392	23.940	1.400	1.398

Table 5.6: Block Cyclic Reduction N=7

N	Q	CPU(parallel)	CPU(seq)	GPU(first)	GPU(second)
7	127	0.002	0.000	0.010	0.007
7	255	0.015	0.020	0.043	0.040
7	511	0.075	0.170	0.170	0.168
7	1023	0.495	1.280	0.390	0.385
7	2047	3.030	9.260	1.050	1.085
7	4095	22.965	70.540	3.795	3.763

Table 5.7: Block Cyclic Reduction N=15

N	Q	CPU(parallel)	CPU(seq)	GPU(first)	GPU(second)
15	127	0.007	0.010	0.020	0.018
15	255	0.035	0.060	0.087	0.085
15	511	0.175	0.410	0.385	0.390
15	1023	1.175	3.080	0.905	0.892
15	2047	7.698	22.800	2.432	2.425
15	4095	53.650	174.750	8.623	8.590

Table 5.8: Block Cyclic Reduction N=31

N	Q	CPU(parallel)	CPU(seq)	GPU(first)	GPU(second)
31	127	0.018	0.020	0.038	0.038
31	255	0.070	0.130	0.185	0.190
31	511	0.365	0.920	0.827	0.820
31	1023	2.392	6.870	1.913	1.910
31	2047	17.355	51.290	5.215	5.215
31	4095	117.445	394.260	18.135	18.083

Table 5.9: Block Cyclic Reduction N=63

N	Q	CPU(parallel)	CPU(seq)	GPU(first)	GPU(second)
63	127	0.040	0.050	0.075	0.072
63	255	0.153	0.280	0.363	0.377
63	511	0.745	1.940	1.690	1.685
63	1023	5.017	14.660	3.973	3.983
63	2047	34.463	109.570	10.705	10.690
63	4095	249.585	844.420	36.375	36.393

Table 5.10: Block Cyclic Reduction N=127

N	Q	CPU(parallel)	CPU(seq)	GPU(first)	GPU(second)
127	255	0.298	0.580	0.757	0.750
127	511	1.530	4.000	3.420	3.425
127	1023	10.340	30.280	7.925	8.038
127	2047	71.278	227.610	21.520	21.497
127	4095	514.878	1753.420	72.642	72.588

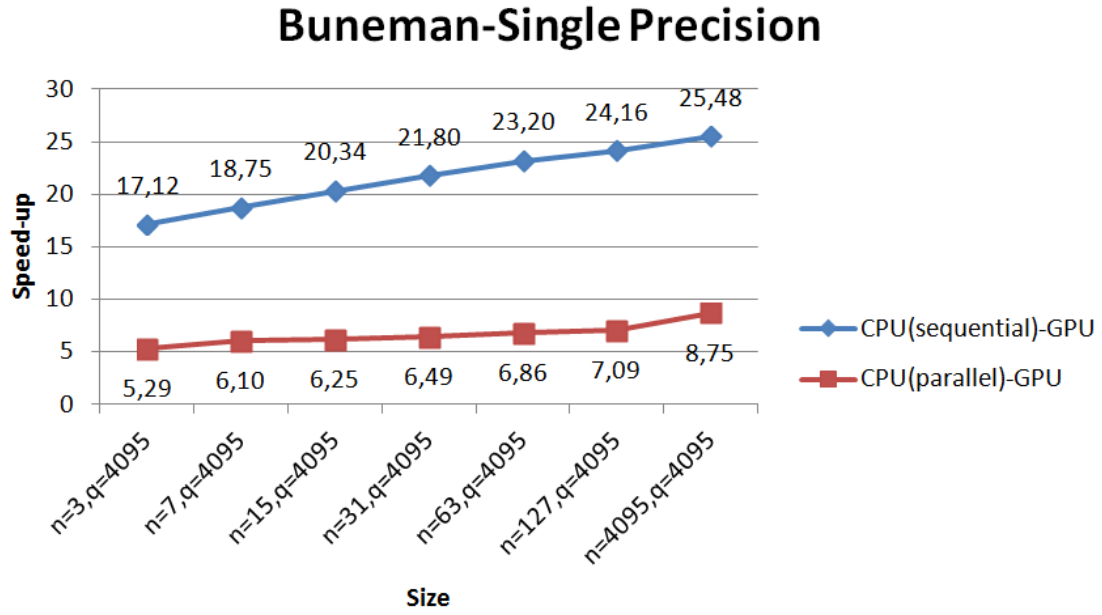


Figure 5-3: Speedup Buneman(SP)

Table 5.11: Block Cyclic Reduction $N=Q$

N	Q	CPU(parallel)	CPU(seq)	GPU(first)	GPU(second)
127	127	0.078	0.100	0.147	0.150
255	255	0.915	1.200	1.487	1.525
511	511	10.177	16.470	13.797	13.778
1023	1023	84.858	252.080	63.520	63.470
2047	2047	1175.078	3802.470	331.875	331.758
4095	4095	20203.910	58816.617	—	2308.483

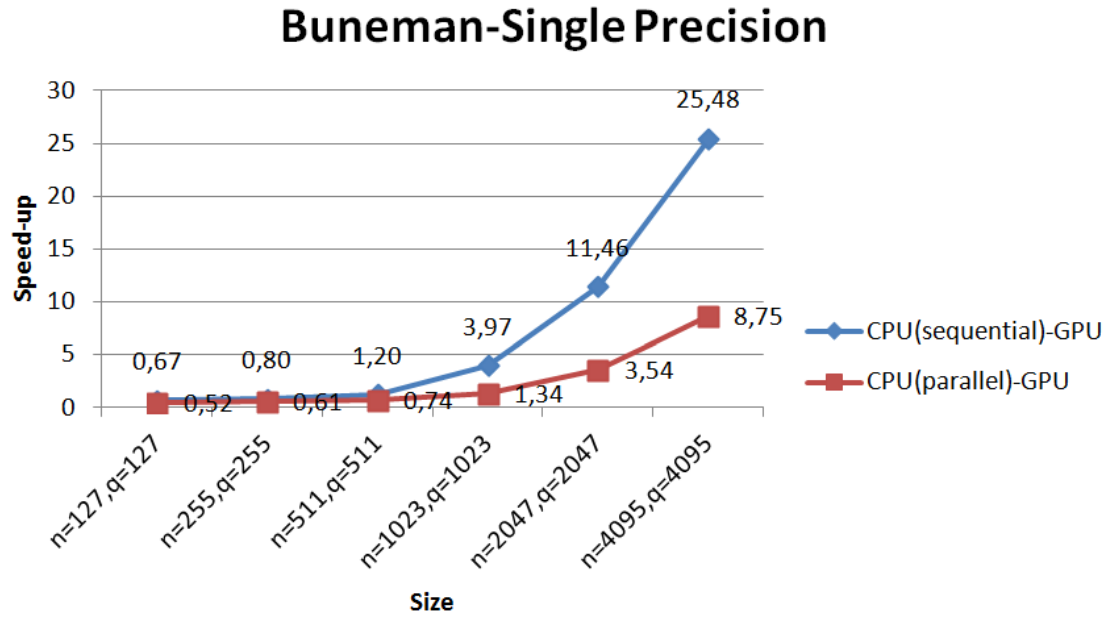


Figure 5-4: Speedup Buneman N=Q (SP)

As can be observed a remarkable speed -up up to 25x can be achieved using GPU instead of sequential CPU. It is worth to be mentioned that as the size increases the speed - up increases but the rate of this growth decreases. The maximum speed-up is achieved when we use the maximum Q size. This is normal because the GPU work load depends on Q size since concerns the inner loop which actually runs on GPU.

5.2.2 Double Precision Performance

Table 5.12: Cyclic Reduction (first version) execution time (DP)

Size	CPU	GPU
2^{21}	0.148	0.033
2^{22}	0.314	0.06
2^{23}	0.619	0.119
2^{24}	1.213	0.236
2^{25}	2.33	0.47
2^{26}	--	--

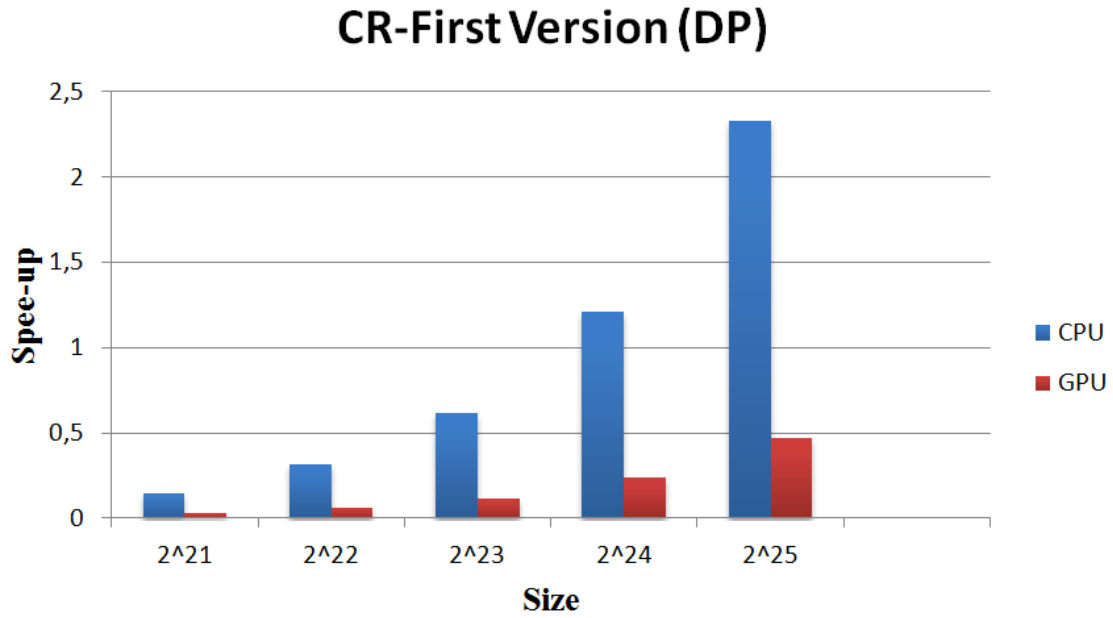


Figure 5-5: CPU-GPU Cyclic Reduction(DP)

Table 5.13: Cyclic Reduction (first version) speed - up (DP)

size	Speed - up
2^{21}	4.48
2^{22}	5.23
2^{23}	5.2
2^{24}	5.13
2^{25}	4.95
2^{26}	—

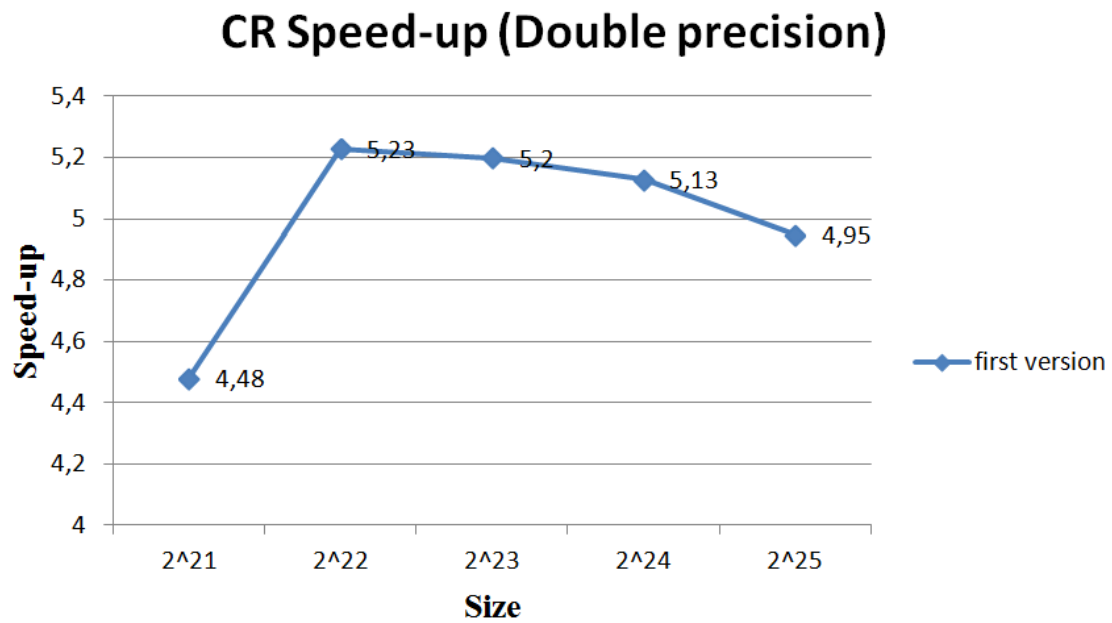


Figure 5-6: Speedup Cyclic Reduction(DP)

Table 5.14: Cyclic Reduction (padding version) execution time (DP)

Size	CPU	GPU
2^{21}	0.156	0.03
2^{22}	0.28	0.06
2^{23}	0.594	0.118
2^{24}	1.09	0.237
2^{25}	2.187	0.463
2^{26}	--	--

Table 5.15: Cyclic Reduction (padding version) speed - up (DP)

size	Speed - up
2^{21}	5.2
2^{22}	4.66
2^{23}	5.03
2^{24}	4.59
2^{25}	4.72
2^{26}	--

Table 5.16: Block Cyclic Reduction N=3

N	Q	CPU(parallel)	CPU(seq)	GPU(first)	GPU(second)
3	127	0.003	0.000	0.010	0.005
3	255	0.013	0.010	0.020	0.018
3	511	0.065	0.120	0.050	0.050
3	1023	0.328	0.820	0.145	0.168
3	2047	2.145	6.200	0.403	0.828
3	4095	17.363	47.700	1.400	4.365

Table 5.17: Block Cyclic Reduction N=7

N	Q	CPU(parallel)	CPU(seq)	GPU(first)	GPU(second)
7	127	0.007	0.010	0.010	0.010
7	255	0.025	0.040	0.035	0.035
7	511	0.145	0.330	0.120	0.117
7	1023	0.948	2.400	0.425	0.422
7	2047	6.697	18.020	2.340	2.325
7	4095	51.395	140.080	10.857	10.825

Table 5.18: Block Cyclic Reduction N=15

N	Q	CPU(parallel)	CPU(seq)	GPU(first)	GPU(second)
15	127	0.013	0.020	0.025	0.022
15	255	0.058	0.110	0.070	0.075
15	511	0.320	0.800	0.268	0.268
15	1023	2.272	5.870	0.993	0.985
15	2047	17.385	44.470	5.300	5.287
15	4095	126.210	347.420	25.312	25.383

Table 5.19: Block Cyclic Reduction N=31

N	Q	CPU(parallel)	CPU(seq)	GPU(first)	GPU(second)
31	127	0.025	0.040	0.040	0.045
31	255	0.115	0.240	0.152	0.150
31	511	0.685	1.760	0.568	0.568
31	1023	5.160	13.110	2.150	2.147
31	2047	32.672	99.650	10.867	10.852
31	4095	243.312	783.830	—	54.682

Table 5.20: Block Cyclic Reduction N=63

N	Q	CPU(parallel)	CPU(seq)	GPU(first)	GPU(second)
63	127	0.050	0.080	0.085	0.087
63	255	0.235	0.510	0.305	0.307
63	511	1.393	3.700	1.167	1.173
63	1023	10.582	27.990	4.503	4.468
63	2047	70.007	213.560	22.192	22.105
63	4095	611.175	1677.440	—	112.907

Table 5.21: Block Cyclic Reduction N=127

N	Q	CPU(parallel)	CPU(seq)	GPU(first)	GPU(second)
127	255	0.475	1.040	0.620	0.625
127	511	2.835	7.630	2.382	2.393
127	1023	21.898	58.090	8.893	8.875
127	2047	149.167	444.920	43.877	43.765
127	4095	1248.453	3493.550	—	231.595

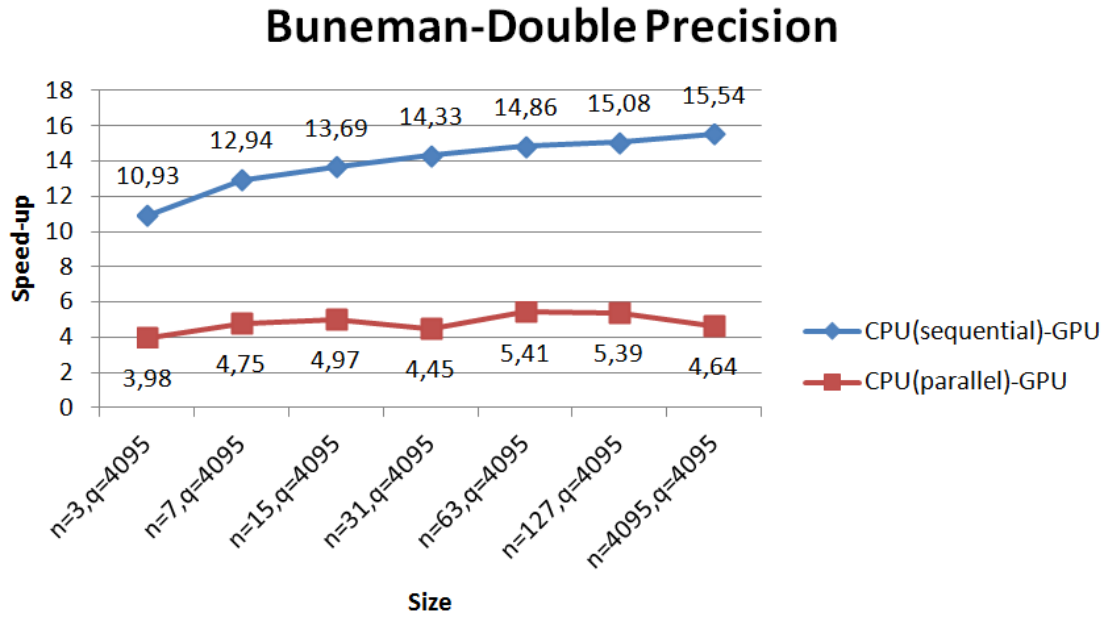


Figure 5-7: Speedup Buneman(DP)

Table 5.22: Block Cyclic Reduction $N=Q$

N	Q	CPU(parallel)	CPU(seq)	GPU(first)	GPU(second)
127	127	0.100	0.160	0.170	0.170
255	255	0.960	2.120	1.250	1.262
511	511	11.355	31.340	9.640	9.697
1023	1023	169.595	482.900	65.748	65.790
2047	2047	,	7437.090	694.060	692.940
4095	4095	,	117200.020	—	7542.910

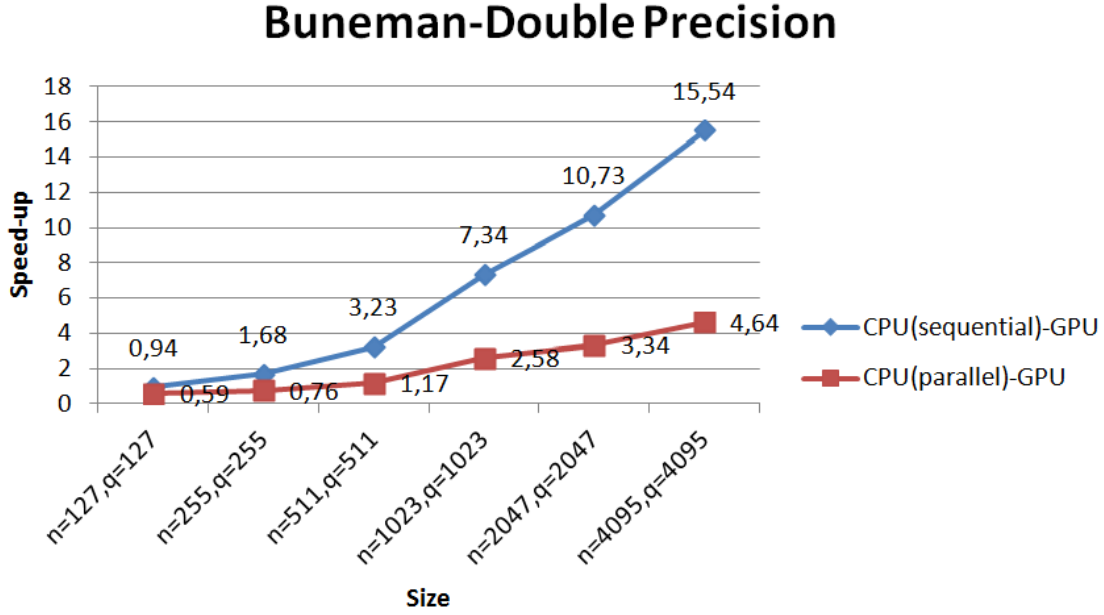


Figure 5-8: Speedup Buneman $N=Q$ (DP)

The same observations with the single precision apply here. In the case of cyclic reduction algorithm the speed - up decrease starts in smaller size due to almost double memory transfers. Likewise, padding does not effect performance. A smaller but equally remarkable speed - up up to 15x was achieved in Block cyclic reduction when ported to GPU. Again the memory transfers now are almost twice therefore consume more significant portion of the total execution time.

5.2.3 Conclusions

Solving PDEs in one dimension and two dimensions demands intensive computation load in both time and space. Due to wide appliance of PDEs in many scientific fields a need for speeding the computation arised. The GPGPU wave came with the solution to this. Particularly, after a deep search in the effort made in this area we conclude that cyclic reduction algorithm was the most efficient to perform on GPUs. Also it is remarkable that very few implementations of block cyclic reduction on GPUs exist and none in our knowledge of Buneman variant.

In cyclic reduction algorithm up to 8x speedup was observed in single precision and up to 5x in double precision. After a certain size speed-up started to decrease due to large memory transfers. Padding technique was applied with no performance impact. In Buneman algorithm up to 25x speedup was observed in single precision and up to 15x in double precision in single threaded CPU and up 8.5x to in single precision and up to 5.5x in double precision. Several optimizations were applied. MKL and Lapack libraries were used for CPU programming and CUBlas and CULapack were used for the GPU programming. In the CPU implementation both sequential and multi- threaded versions were made. A sliding window technique was used in order to increase the problem size. An attempt to overlap computations with memory tranfers using streams was made with no performance impact. In both algorithms single precision outperforms double precision since in the second approximately twofold memory tranfers occurred. Conclusively, it was proven that solving PDEs in both one and two dimensions on GPUs can give significant performance in relation to CPU implementation.

5.2.4 Further Improvements

As referred in previous sections both algorithms were tested using the Poisson differential equation as a model. However, the implementations were implemented in a way to be easily expanded in order to model several mathematical and physical problems. As a case study we plan to apply the Buneman method to a reaction-diffusion PDE. This PDE is considered as a common simplified model for studying the expansion of gliomata in the human brain. This would lead to adapt time as a parameter in the calculation process. Buneman algorithm will be called repeatably in a loop, which will calculate the PDE's coefficients in every time step.

Appendix A

Bibliography

[1]Nvidia, “Kepler GK110, The Fastest, Most Efficient HPC Architecture Ever Built” [Whitepaper]. <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>

[2]Christos D. Antonopoulos, “High performance computing” [Lecture Notes], University of Thessaly, 2013. <http://inf-server.inf.uth.gr/courses/CE421/lectures.php>

[3]Nvidia, “CUDA Programming Model Overview”. <http://moss.csc.ncsu.edu/~mueller/-cluster/nvidia/GPU+CUDA.pdf>

[4]Nvidia, “CUDA C Programming Guide”. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>

[5]T. Brandvik and G. Pullan . ”SBLOCK: a framework for efficient Stencil-Based PDE solvers on multi-core platforms”, In Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on, page 1181–1188, 2010

[6]P. Micikevicius. 2009. ”3d Finite difference computation on GPUs using CUDA,” GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (New York, NY, USA), ACM, 2009

[7]David L. Foster. 2011,”GPU Acceleration of Solving Parabolic Partial Differential Equations Using Difference Equations”: Proceedings of the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications, 2011

[8]S. Williams, J. Shalf, L. Oliker, S. Kamil , P. Husbands, and K. Yelick, “Scientific Computing Kernels on the Cell Processor,” Int. J. Parallel Program., vol. 35, no. 3, pp. 63–298, 2007

[9]Schäfer, Andreas and Fey, Dietmar. (2011) ”High Performance Stencil Code Algorithms for GPGPUs”: Proceedings of the International Conference on Computational Science, ICCS 2011

[10]D.P. Playne and K.A. Hawick. (2010) ”Hierarchical and Multi-level Schemes for Finite Difference Methods on GPUs ”: Technical Note, Computer Science, Massey University

[11]Daniel Egloff, GPUs in Financial Computing Part I: High-Performance Tridiagonal Solvers on GPUs. Wilmott Magazine, September 1, 2010

[12]Daniel Egloff, GPUs in financial computing part III: ADI solvers on GPUs with application to stochastic volatility, Wilmott Magazine, March 2011

[13]Won-Ki Jeong, Ross Whitaker, Mark Dobin. (2006) “Interactive 3D Seismic Fault Detection on the Graphics Hardware.”: Proceedings of International Workshop on Volume Graphics 2006, pp.111-118

[14]Hee-Seok Kim, Shengzhao Wu, Li-wen Chang and Wen-mei W. Hwu “A Scalable Tridiagonal Solver for GPUs” Electrical and Computer Engineering ,University of Illinois at Urbana-Champaign

[15]R. S. Oliveira, B. M. Rocha, R. M. Amorim et al., “Comparing Cuda, OpenCL and OpenGL implementations of the cardiac monodomain equations,” in Parallel Processing and Applied Mathematics, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds., vol. 7204 of Lecture Notes in Computer Science, pp. 111–120, Springer, Berlin, Germany, 2012

[16]Jens Kruger and Rudiger Westermann, “Linear Algebra Operators for GPU Implementation of Numerical Algorithms” International Conf. on Computer Graphics and Interactive Techniques, 2005

[17]Daisuke Sato , Yuanfang Xie , James N. Weiss ,Zhilin Qu ,Alan Garfinkel, Allen R. Sanderson ,”Acceleration of cardiac tissue simulation with graphic processing units”, 2009

[18]Andrés Mena and Jose F Rodriguez, ”Using Graphic Processor Units for the Study of Electric Propagation in Realistic Heart Models”, Computing in Cardiology (CinC) Publication Year: 2012, Page(s): 37 - 40

[19]Li-Wen Chang, John A. Stratton, Hee-Seok Kim, and Wen-Mei W. Hwu ,”A Scalable, Numerically Stable, High-performance Tridiagonal Solver using GPUs”, University of Illinois at Urbana-Champaign

[20]Abhijeet Gaikwad and Ioane Muni Toke, ” Parallel Iterative Linear Solvers on GPU: A Financial Engineering Case ” , In proceeding of: Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP 2010, Pisa, Italy, February 17-19, 2010

[21]Section of Scientific Computing ,Department of Informatics and Mathematical Modeling, Technical University of Denmark ,”GPULab Library - a High-Performance Library for PDE Solvers”

[22]Jeff Bolz, Ian Farmer, Eitan Grinspun ,Peter Schröder,” Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid”

[23]Dominik Göddeke, Robert Strzodka: “Cyclic Reduction Tridiagonal Solvers on GPUs Applied to Mixed-Precision Multigrid”. IEEE Trans. Parallel Distrib. Syst. 22(1): 22-32 (2011)

[24]Y. Zhang, J. Cohen, and J. D. Owens, “Fast tridiagonal solvers on the GPU,” in Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010), Jan. 2010, pp. 127–136

[25]D. Egloff, “High performance finite difference PDE solvers on GPUs.” Available at: http://download.quantalea.net/fdm_gpu.pdf, Feb. 2010

[26]R. W. Hockney, A fast direct solution of Poisson’s equation using Fourier analysis, J. ACM, 12 (1965)